Josef Pöpsel
Ute Claussen
Rolf-Dieter Klein
Jürgen Plate

Computergrafik

Algorithmen und Implementierung

mit Pascal-Programmen

Springer-Verlag

Josef Pöpsel
Ute Claussen
Rolf-Dieter Klein
Jürgen Plate

Computergrafik

Algorithmen und Implementierung

Mit 133 Abbildungen, 16 Farbtafeln,
17 Tabellen und 1 Diskette

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Josef Pöpsel
Frohlinder Straße 46
44577 Castrop-Rauxel-Schwerin

Ute Claussen
Frohlinder Straße 46
44577 Castrop-Rauxel-Schwerin

Rolf-Dieter Klein
Straßberger Straße 34
80809 München

Jürgen Plate
Riedererstraße 13
85737 Ismaning

Die Deutsche Bibliothek – CIP-Einheitsaufnahme.
Computergrafik: Algorithmen und Implementierung / Josef Pöpsel . . . – Berlin; Heidelberg; New York; London; Paris; Tokyo; Hong Kong; Barcelona; Budapest: Springer, 1994

NE: Pöpsel, Josef

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes der Bundesrepublik Deutschland vom 9. September 1965 in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechtsgesetzes.

© Springer-Verlag Berlin Heidelberg 1994
Softcover reprint of the hardcover 1st edition 1994

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, daß solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Umschlaggestaltung: Konzept & Design, Ilvesheim
Satz: Mit TEX erstellte reproduktionsfertige Vorlage von den Autoren

ISBN-13: 978-3-642-46800-1 e-ISBN-13: 978-3-642-46799-8
DOI: 10.1007/978-3-642-46799-8

Für meine Mutter Hildegard, der ich dieses Buch gerne noch gezeigt hätte.

Josef

Vorwort

Was lange währt, wird endlich gut.

Normalerweise bezieht sich ein Vorwort auf die Mühen und Umstände bei der Formulierung und der Produktion eines Buches. Die Autorenschaft entschuldigt sich beim Verlag für alle eingetretenen Verzögerungen und bedankt sich für die Geduld der beteiligten Menschen. In diesem Fall könnte allein das Vorwort den Umfang eines Kapitels annehmen. Es wäre dennoch längst nicht alles an Umständen ausgedrückt.

Die Autoren und die Autorin, alle erfahren in der Schaffung von umfangreichen Werken, mußten bei dem Versuch, dieses Buch zu veröffentlichen, Widerstände überwinden, die außerhalb einer fachlichen Auseinandersetzung lagen. Die Zeit, die darüber vergangen ist, hat uns an Erfahrungen reicher gemacht und geholfen, das Buch in seiner Qualität zu verbessern.

Die Liste der Menschen, bei denen wir uns für ihre Hilfe bedanken, wäre also um eine Liste der Personen und Institutionen zu ergänzen, die uns in dem Ziel behindert haben, das Buch zu veröffentlichen. Wir verzichten darauf.

An erster Stelle gilt unser Dank dem Springer-Verlag und insbesondere Herrn Dr. Michael Barabas. Er hat sich für die Herausgabe des Buches engagiert und sich von unserer „Redlichkeit" überzeugen lassen. Gemeinsam mit seinen Kolleginnen und Kollegen, Frau Glaunsinger, Herrn Rossbach und Frau Malachowski ist er für uns eine fachliche und menschliche Unterstützung.

Prof. Heinrich Müller von der Universität Dortmund stellte uns seine Fachkompetenz weit über den kollegialen Rahmen hinaus zur Verfügung. Seine Antrittsvorlesung in Dortmund gab indirekten Anstoß für das Weiterverfolgen der Veröffentlichung.

Die von den Streitigkeiten direkt betroffenen Autoren bedanken sich bei den anderen beiden Autoren für ihre Solidarität und den Geduld im Umgang miteinander.

Castrop-Rauxel, im Juli 1993 — *Ute Claussen*
Josef Pöpsel
München — *Rolf-Dieter Klein*
Jürgen Plate

Inhaltsverzeichnis

1. Einleitung 1
1.1 Überblick über den Inhalt 2
1.2 Wie können Sie dieses Buch benutzen? 3
1.3 Literatur 4
1.4 Sie sind gefragt 5

I 2D-Algorithmen

2. Koordinatensysteme 9
2.1 Koordinatentransformation 10
2.2 Weltkoordinaten und Bildkoordinaten 10
2.3 Skalierung und Fenster 10
2.4 Transformationen 12
2.5 Polarkoordinaten 16
2.6 Homogene Koordinaten 17
2.7 Zusammensetzen von Transformationen 19

3. Algorithmen 21
3.1 Abschneiden (Clipping) und Überdecken 21
3.2 Polygon-Clipping 24
3.3 Flächenfüllen und Schraffieren 24
3.3.1 Flächenfüllen am Bildschirm 25
3.3.2 Schraffieren 26

4. Anwendungen der 2D-Grafik 29
4.1 Darstellen von Funktionen und Wertetabellen 29
4.2 Lineare Approximation 30
4.3 Logarithmische Approximation 32
4.4 Exponentielle Approximation 32
4.5 Gleitender Durchschnitt 32
4.6 Polynom-Interpolation 33
4.7 Spline-Interpolation 34
4.8 Bézier-Kurven 36
4.9 Stereobilder 37

II Grundlagen 3D-Algorithmen und Rendering

5. Mathematische Grundlagen 41
5.1 Punkte und Vektoren 41
5.2 Matrizen 43
5.3 Vektor- und Matrizenalgebra 43
5.4 Transformationen 45
5.4.1 Transformation von Punkten und Vektoren 46
5.4.2 Transformation von Normalenvektoren 47
5.4.3 Transformation von Koordinatensystemen 48
5.4.4 Projektionen 48

6. Beschreibung der Programme 51
6.1 Möglichkeiten der Grafikprogrammierung 51
6.2 Prinzipielles zur Implementierung 52
6.3 Die Bedienung des Hauptprogramms `PRIM` 53
6.4 Beispiele zu den mathematischen Grundlagen 53

7. Grundlagen der dreidimensionalen Computergrafik 57
7.1 Koordinatensysteme 57
7.2 Modellierung 62
7.3 Entfernung der Rückflächen 65
7.4 Clipping 66
7.5 Verdeckungsrechnung 67
7.5.1 Der Maler-Algorithmus 68
7.5.2 Der Tiefensortier-Algorithmus 68
7.5.3 Der z-Buffer-Algorithmus 68
7.5.4 Scanline-Algorithmen 69
7.6 Beleuchtungs- und Schattierungsverfahren 69
7.7 Scankonvertierung 69
7.8 Antialiasing 71

8. Beispiele zu den Grundlagen der 3D-Grafik 73
8.1 Modellierung und Transformation zwischen Koordinatensystemen 73
8.2 Weitere Koordinatensysteme 76
8.3 Clipping 76

9. Implementierung der grundlegenden Algorithmen 79
9.1 Die von `PRIM` verwendeten Units 79
9.2 Die Unit `PRIMVIEW`, Matrizenrechnung zum Anfassen 79
9.3 Die Unit `PRIMREND`, die Rendering-Pipeline aus Programmsicht 82

10. Modelle für die Beleuchtungsrechnung 89
10.1 Farben 89
10.2 Lichtquellen 90
10.3 Abschwächung der Intensität durch Entfernung 93

10.4 Abschwächung mit Nebeleffekt 94
10.5 Abschwächung der Intensität in Medien 95
10.6 Oberflächen 95

11. Beispiele zur Beleuchtungsberechnung 99
11.1 Farbmodelle 99
11.2 Lichtquellenmodelle 100
11.3 Oberflächenmodelle 101

12. Implementierung der Beleuchtungsmodelle 105
12.1 Datenstrukturen 105
12.2 Algorithmen 107

13. Darstellungsmethoden (Rendering) 113
13.1 Verdeckungsrechnung mit dem z-Buffer-Verfahren 113
13.2 Scankonvertierung von Dreiecken und Polygonen 114
13.3 Linear interpolierende Schattierungsverfahren 115
13.4 Texturen 117
13.4.1 Farbtextur-Mapping 118
13.4.2 Bumpmapping 119
13.4.3 Volumentexturen 120
13.5 Transparenz 121

14. Beispiele zu den Darstellungsmethoden 123
14.1 Schattierungsverfahren 123
14.2 Texturierung 124

15. Implementierung der Darstellungsmethoden 127
15.1 Gemeinsamkeiten der Darstellungskonzepte 127
15.2 Interpolationen 129
15.3 Rasterisierung und Schattierung von Dreiecken 131
15.3.1 Rasterisierung von Dreiecken 131
15.3.2 Bestimmung der linken Span-Anfangspunkte 133
15.3.3 Bilineare Interpolation in einem Dreieck 140
15.4 Die Implementierung von Schattierung und Rasterisierung . . 142
15.4.1 Die Prozedur `SetzeSchattierungsParameter` 143
15.4.2 Die Prozedur `StelleFlaecheDar` 144
15.4.3 Die Prozedur `ZeichneDreieck` 149
15.4.4 Die Prozedur `ZeichneHintergrund` 149

III Radiosity

16. Radiosity 153
16.1 Berechnung der Formfaktoren 156
16.2 Lösung des Gleichungssystems 159
16.3 Mehrpassverfahren 160

17. Beispiele zur Radiosity-Berechnung 161

18. Implementierung des Radiosity-Verfahrens 165
18.1 Lösung des Gleichungssystems 165
18.1.1 Iteratives Lösungsverfahren 167
18.2 Berechnung der Formfaktoren 169
18.3 Struktur des implementierten Verfahrens 169
18.3.1 Vor- und Nachteile des iterativen Verfahrens 171
18.4 Die Unit `PRIMRAD`, Radiosity zum Anfassen 172
18.4.1 Die Prozedur `ErzeugePatchesUndElemente` 173
18.4.2 Nachbarschaften 175
18.4.3 Die Prozedur `BerechneFlaechenGroessenUndAmb` . . . 180
18.4.4 Die Prozedur `BerechnedFFs` 181
18.4.5 Die Prozedur `SuchePatchMitGroesstemdBA` 181
18.4.6 Die Prozedur `BestimmeMatrizenFuerSingleplane` . . 182
18.4.7 Die Prozedur `ElementeAufSingleplane` 183
18.4.8 Die Prozedur `BerechneFormfaktoren` 184
18.4.9 Die Prozedur `WandeleFormfaktorenInEnergieUm` . . . 185
18.4.10 Die Prozeduren `MittlereBeteiligteFlaechen` und `BerechnePunktFarbenAusEnergien` 187

IV Raytracing

19. Raytracing 191
19.1 Die Strahlverfolgung 191
19.1.1 Einführung 191
19.1.2 Lichtstrahlen 194
19.1.3 Schatten und Spiegelungen 197
19.1.4 Brechungen 199
19.1.5 Der Raytrace-Algorithmus 201
19.2 Grundkörper 207
19.2.1 Kugeln 207
19.2.1.1 Schnitt: Kugel und Strahl 207
19.2.1.2 Texturen und Kugeln 212
19.2.2 Quadriken 213
19.2.2.1 Ellipsoid 214
19.2.2.2 Hyperboloid 215
19.2.2.3 Kegel 216
19.2.2.4 Elliptischer Zylinder 217
19.2.2.5 Implementierung für den symmetrischen Kegelstumpf 219
19.2.3 Ebenen und flächige Objekte 229
19.2.3.1 Schnitt: Ebene und Strahl 229
19.2.3.2 Schnitt: Kreisfläche und Strahl 231
19.2.3.3 Schnitt: konvexes Viereck und Strahl 232

19.2.3.4 Schnitt: Dreieck und Strahl 235
19.2.3.5 Schnitt: Polygon und Strahl 236
19.2.4 Rotationskörper 237
19.3 Konstruktive Geometrie und Raytracing: CSG 239
19.3.1 Der CSG-Algorithmus 243
19.4 Verbesserte Raytracing-Techniken 246
19.4.1 Supersampling 246
19.4.2 Adaptives Supersampling 247
19.4.3 Stochastic Sampling und Distributed Raytracing 249
19.4.3.1 Halbschatten 249
19.4.3.2 Bewegungsunschärfe (Motion Blur) 251
19.5 Beschleunigung von Raytracing-Verfahren 252
19.5.1 Bounding-Box-Algorithmen 252
19.5.2 Parallelisierung 255
19.6 Public-Domain-Software 255

V Hardware

20. Grafik-Hardware 259
20.1 Aufbau und Wirkungsweise 259
20.1.1 Der Bildwiederholspeicher 259
20.1.2 Die Farbpalette 261
20.1.2.1 Setzen eines einzelnen Farbregisters 264
20.1.2.2 Setzen eines Blocks von Farbregistern 265
20.1.2.3 Programmbeispiel 266
20.1.3 Animation mit Farbpaletten 266
20.1.4 Nachbildung von True-Color 268
20.1.4.1 Halbtontechnik 268
20.1.4.2 Ordered Dither nach Bayer 270
20.1.4.3 Floyd-Steinberg 275
20.2 PC-Grafikkarten 279
20.2.1 Super-VGA auf dem PC 279
20.2.1.1 Bank Select bei ET4000 283
20.2.1.2 Bank Select bei VESA-Karten 285
20.2.2 HiColor-VGA-Karten 291
20.2.3 True-Color-Karten 293
20.3 3D-Darstellung in Stereo 295
20.3.1 Rot-Grün-Brille 295
20.3.2 LCD-Shutter-Brille 299
20.4 Grafik-Beschleunigerkarten 305
20.4.1 Beschleunigerkarten mit S3 306
20.4.1.1 Hardware-Cursor 306
20.4.1.2 Grafikbefehle 306
20.4.2 Aufbau der DSM860-Beschleunigerkarte 313
20.4.2.1 Aufbau des Intel i860 Prozessors 314

20.4.2.2 Die Einheiten des i860 314
20.4.2.3 Register und Befehle 317
20.4.2.4 Die Befehle für die Grafik-Recheneinheit . . . 320
20.4.2.5 Zugriff auf den i860-Bildspeicher 321

VI Anhang: Programmierung von Rendering und Radiosity

A. Beschreibung der Eingabesprache GED 325
A.1 Die Eingabesprache GED 325
A.2 Objekte 327
A.3 Teilobjekte 328
A.3.1 Polygonale Teilobjekte 329
A.3.2 Referenzen auf Objekte 329
A.3.3 Vordefinierte Körper 329
A.3.4 CSG-Ausdrücke 330
A.3.5 Transformationen und Oberflächeneigenschaften . . . 330
A.3.6 Teilobjekte im Detail: Polygonobjekte 331
A.3.7 Teilobjekte im Detail: Vordefinierte Körper 334
A.3.7.1 Die Kugel als vordefinierter Körper 334
A.3.7.2 Der Würfel als vordefinierter Körper 334
A.3.7.3 Der Zylinder als vordefinierter Körper 336
A.3.7.4 Der Kegel als vordefinierter Körper 336
A.3.7.5 Der Torus als vordefinierter Körper 336
A.4 Transformationen 338
A.5 Oberflächeneigenschaften 338
A.6 Gültigkeitsbereiche 340
A.7 Angaben direkt vor der Szene 340
A.7.1 Angaben zum Betrachter 341
A.7.2 Angaben zu den Lichtquellen 342
A.8 Hintergrund 342
A.9 Die von LeseGEDDatei erzeugten Datenstrukturen 345
A.9.1 Die Unit PRIMGLOB 345
A.9.1.1 Zeigertypen von PRIMGLOB 345
A.9.1.2 Abspeichern von Objekten 347
A.9.1.3 Abspeichern des Betrachters 348
A.9.1.4 Abspeichern von Lichtquellen 348
A.9.1.5 Abspeichern des Hintergrundes 348
A.9.1.6 Abspeichern von polygonalen Teilobjekten . . 349
A.9.1.7 Abspeichern von CSG-Ausdrücken 351
A.9.1.8 Abspeichern von Oberflächeneigenschaften . . 352
A.9.1.9 Aufzählungstypen und allgemeine Typen . . . 353
A.9.1.10 Der GlobaleFlaecheTYP 354
A.10 Die Units PRIMREAD und PRIMDERE 356
A.10.1 Die Prozedur LeseGEDDatei 356

A.10.2 Die Prozedur `DereferenziereSzene` 357
A.11 Format der Texturdateien 358

B. Syntax der Grafischen Eingabesprache 361
B.1 Allgemeine Festlegungen 361
B.2 Die Syntax der Beschreibung 361
B.3 Sprachdefinition 362
B.4 Liste der reservierten Worte 365

C. Hardwareabhängige Module von `PRIM` 367
C.1 Ansprechen des erweiterten Speichers 368
C.2 Die Routinen von `PRIMMEM` 371
C.2.1 Routinen zum Ansprechen des z-Buffers 371
C.2.2 Routinen zum Ansprechen des item-Buffers 373
C.2.3 Routinen zum gemeinsamen Ansprechen von z- und item-Buffer 374
C.2.4 Routinen zum Ansprechen des Deltaformfaktor-Buffers 375
C.2.5 Routinen zum Ansprechen von Texturen 375
C.2.6 Routinen zum Ansprechen von Speicher für polygonale Teilobjekte 377
C.2.7 Routinen zum Ansprechen der Punkt- und Flächenliste 378
C.2.8 Routinen zum Ansprechen von Formfaktoren und unverteilter Energie einer Fläche 380
C.3 Die Routinen von `PRIMOUT` 381

Literaturverzeichnis 385

Index 389

1. Einleitung

Bücher zu Computergrafik finden bei Leserinnen und Lesern immer größeren Anklang. Mittlerweile besitzt ein Großteil der PC-Benutzer einen Farbmonitor sowie eine VGA-Karte mit relativ hoher Auflösung. Auch „Otto Normalverbraucher" steht nichts mehr im Wege, Grafiken hoher Qualität auf dem Rechner zu erzeugen.

Was fehlt, sind einerseits die nötigen Kenntnisse über die Verfahren der Computergrafik und andererseits die Möglichkeit, das Ganze auch ausprobieren zu können. Beide Punkte versuchen wir, mit diesem Buch zu verändern. Sowohl Neulinge als auch Spezialisten werden hier Methoden für sich finden.

Ausgehend von den Grundlagen zweidimensionaler Bildgenerierung werden die Aspekte dreidimensionaler Darstellung und die damit verbundenen Probleme vertieft. Die detaillierte Darstellung der derzeit am häufigsten verwendeten Darstellungsmethoden, des Rendering, des Radiosity- und des Raytracing-Verfahrens in Text und Programm ermöglichen einen intensiven Einblick. Mit den beigelegten Programmen können auf einem PC Bilder mit Hilfe einer einfachen Eingabesprache generiert werden, die sonst das Verständnis eines kompletten Programmsystems voraussetzten. Viele Hinweise zur Programmierung auf dem PC und den damit verbundenen Anforderungen schaffen die Grundlage für eigene Erweiterungen der Software.

Die für dieses Buch speziell entwickelte Rendering- und Radiosity-Software ist, neben vielen exemplarischen Beispielen von der Farbtabellenanimation bis hin zur Bedienung der Stereobrille, eine ausgereifte, „frei" benutzbare Software. Bis auf zwei Ausnahmen wurden alle farbigen Abbildungen des Buches mit der beigelegten Software erzeugt. Bei der Implementierung haben wir versucht, auf eine möglichst einfache Portierung zu achten. Alle Schnittstellen sind offen, die Algorithmen im Detail im Buch beschrieben. Die Software für Rendering und Radiosity einerseits und Raytracing andererseits basiert nicht auf der gleichen Eingabe, ist aber für eine Integration vorbereitet. Wir möchten es den Leserinnen und Lesern überlassen, dies selbst zu vollziehen. Dabei eignen Sie sich die Grundlagen der Verfahren dann „automatisch" an.

1.1 Überblick über den Inhalt

Computergrafik – die „bunteste“ Form der Informationsverarbeitung umfaßt drei Bereiche: Die Bildanalyse, also das Erkennen von Bildinformation, die Bildgenerierung, das Erzeugen von Bildinformation und die Bildverarbeitung, die von vielen aber nicht mehr als Gegenstand der Computergrafik betrachtet wird. Wir wollen uns in diesem Buch auf das Generieren von Bildern konzentrieren.

Teil 1 ist daher ein einführender Teil zur zweidimensionalen Computergrafik und den mathematischen Grundlagen. Folgende Fragen „Wie werden auf dem Monitor Linien dargestellt?“ oder „Wie schraffiere ich eine Fläche?“ sind Gegenstand dieses ersten Teils.

Der zweite Teil befaßt sich mit den mathematischen und algorithmischen Grundlagen der dreidimensionalen Computergrafik. Hier wird der Übergang von der „flachen“ 2D-Welt in die räumliche Welt vollzogen. „Wie wird eine gedachte 3D-Welt auf einen Bildschirm abgebildet?“ und „Wie ist die Modellierung dreidimensionaler Welten möglich?“ sind nur zwei der vielen Fragen, die in diesem Teil beantwortet werden.

Teil 3 des Buches befaßt sich dann mit der mehr oder weniger guten Simulation der wahrnehmbaren Welt. Lichtquellen, Oberflächeneigenschaften, Farbe, Reflexion sind nur einige der Stichpunkte, die dieser Teil behandelt. Wir gehen sehr stark auf die Frage „Wie verteilt sich Licht im Raum?“ ein, denn sie bildet die Grundlage für das Verständnis der folgenden Abschnitte. Eines der aktuellsten Forschungs- und Entwicklungsgebiete der Computergrafik – das Radiosity-Verfahren – wird hier detailliert dargestellt.

Der folgende Teil ist dem Raytracing gewidmet. Allerorts trifft man heute auf spiegelnde Kugeln, „flying logos“ o.ä. Meistens sind solche Bilder mittels Raytracing berechnet. Vom grundlegenden Algorithmus, der sehr einfach zu verstehen ist, über die Frage „Wo schneidet ein Strahl einen Kegelstumpf und wie berechne ich diesen Schnittpunkt?“ bis zur Darstellung von Bewegungsunschärfe geht das Spektrum der dargestellten Methoden.

Die Teile 3 (Radiosity) und 4 (Raytracing) des Buches sind relativ unabhängig voneinander. Für beide ist allerdings die Kenntnis der dreidimensionalen mathematischen Grundlagen (Teil 2) und teilweise der 2D-Grundlagen notwendig. Meist können die 3D-Grundlagen durch Verallgemeinerung der 2D-Methoden gewonnen werden.

Ohne Hardware nutzt die beste Computergrafik-Software nichts. Nicht zuletzt aus diesem Grund haben wir der Grafik-Hardware den letzten Teil dieses Buches gewidmet. Hier werden wichtige Begriffe der Hardware, z.B. der „Bildwiederholspeicher“ erläutert. Anhand ausgesuchter Komponenten, die breit verfügbar sind, wird das Ansprechen und die Steuerung von Grafikkarten und Beschleunigerkarten erklärt.

Natürlich können in einer solchen aktuellen Darstellung nicht alle möglichen verfügbaren Komponenten berücksichtigt werden, eine Auswahl schließt auch aus. Verstehen Sie deshalb die Diskussion als exemplarische Darstellung.

Ausführliche Anhänge geben Einblick in die beiliegende Software. Sie sind für diejenigen Leserinnen und Leser gedacht, die ganz genau wissen wollen, wie es funktioniert. Im Anhang A finden Sie eine detaillierte Darstellung der grafischen Eingabesprache GED und wie mit ihrer Hilfe komplexe Objekte modelliert werden. Ein zweiter Abschnitt faßt die Syntax dann nochmals zusammen. Da sich die Implementierung von Software nicht vollständig hardwareunabhängig gestalten läßt, werden im letzten Kapitel des Anhangs diese hardwareabhängigen Teile diskutiert.

Parallel zu den theoretischen Inhalten werden mit Beispielen aus der Software die einzelnen Effekte, Methoden und Verfahren demonstriert. So lernen Sie die Software und ihre Möglichkeiten nach und nach kennen.

1.2 Wie können Sie dieses Buch benutzen?

Sie können das Buch auf verschiedene Art und Weise nutzen: als Lehr- und Lernbuch, als Nachschlagewerk oder zur Unterstützung eigener Projektarbeit im Bereich Computergrafik. Integraler Bestandteil ist dabei die Software, die dem Buch beiliegt.

Zu Ihrer Unterstützung beim Lernen und Lesen wurden unterschiedliche Darstellungen für unterschiedliche Methoden gewählt. Algorithmen, die sprachunabhängig Verfahren beschreiben, werden mit einer *geneigten Schreibschrift* gesetzt und mit fortlaufenden Nummern versehen, wie z.B. der Algorithmus für das „flat shading“:

Algorithmus 1.1 (Flat Shading)

WENN $Z_i < z\text{-}Buffer[x, y]$, *DANN:*
  $z\text{-}Buffer[x, y] := Z_i$.
  $FrameBuffer[x, y] :=$ *Farbe des Polygons.*
SONST tue nichts.

Mathematische Formeln und Bezeichnungen, wie beispielsweise $x^2 + y^2$ sind ebenfalls geneigt gesetzt.
Zitate aus den beiliegenden Pascal-Programmen werden in `Schreibmaschinenschrift` dargestellt, z.B. die in einem bestimmten Modul vereinbarten Funktionsnamen:

```
PROCEDURE SetzeSchattierungsParameter(..)
PROCEDURE StelleFlaecheDar(..)
PROCEDURE ZeichneHintergrund(..)
PROCEDURE ZeichneDreieck(..)
```

Dabei sind Schlüsselworte der Programmiersprache zur leichteren Erkennbarkeit groß geschrieben. Im laufenden Text eingestreute Variablen- oder Prozedurnamen sind, so wie auch abgesetzte Programmtexte, ebenfalls mit Schreibmaschinenschrift gestaltet.

Die einzelnen Teile des Buches sind so aufgebaut, daß ein Durcharbeiten von vorne nach hinten möglich ist. Haben Sie keine oder nur geringe Kenntnisse

der Computergrafik, so sollten Sie den ersten und den zweiten Teil gründlich bearbeiten. Für Fragen, die hier offen bleiben, sind vertiefende Literaturangaben am Ende dieser Einleitung und im fortlaufenden Text zu finden. Die beiliegende Software können Sie während der Bearbeitung zum Experimentieren und Anwenden benutzen. Im Text werden Beispiele ausführlich diskutiert.

Sind Sie bereits fortgeschritten in den Kenntnissen der Computergrafik und wollen sich mit einem der Spezialgebiete Raytracing, Radiosity oder Hardware auseinandersetzen, so können Sie direkt die entsprechenden Teile bearbeiten und nur bei eventuellen Rückfragen auf die ersten beiden zurückgreifen. Für diese Leserinnen und Leser sind wahrscheinlich auch die Kapitel „Implementierung von ... " interessant. Sind Sie nur an der Anwendung interessiert, so können Sie diese Abschnitte überspringen.

Der Teil zum Thema Hardware und die Anhänge sind dann geeignet, wenn sie selbst Computergrafik programmieren oder unsere Programme auf Ihren Rechner anpassen wollen. Respektieren Sie dabei – bei aller Offenheit – die gültigen Bestimmungen bezüglich des Einsatzes und der Verwendung von Software!

1.3 Literatur

Dieses Buch kann nicht alle Fragen zur Computergrafik beantworten. Je nachdem, wie Ihr Interesse an diesem Thema ist, werden Sie Originalliteratur, weitere Lehrbücher oder vertiefende Bücher benötigen. Wir erläutern aus dem bestehenden Angebot hier nur einige wenige, in denen Sie dann natürlich auch wieder weitere Hinweise finden.

1. Foley, van Dam, Feiner, Hughes: *Computer Graphics – Principles and Practice*, Addison-Wesley, 1990 [FvDFH90].
   Dieses Buch ist eigentlich konkurrenzlos in der Breite und Aktualität der Darstellung, da es alle Bereiche der Computergrafik abdeckt. Es hat allerdings drei wesentliche Nachteile: es ist relativ teuer, in Englisch geschrieben und mit 1200 Seiten extrem umfangreich. Es enthält keine Software. Alle Themen, die wir hier behandeln, sind in diesem Grundlagenwerk enthalten und werden mit vielen zusätzlichen historischen und aktuellen Verfahren dargestellt. Der Übergang von der Theorie zur Praxis fehlt allerdings an vielen Stellen.
2. Watt, Watt: *Advanced Animation and Rendering Techniques – Theory and Practice*, Addison-Wesley, 1992 [WW92].
   Eine sinnvolle Ergänzung in Richtung praktischer Anwendbarkeit ist das Buch von Vater und Sohn Watt. Die aktuell ausgewählten Algorithmen sind teilweise auch in der Programmiersprache C abgedruckt. Alle Verfahren werden sowohl theoretisch als auch praktisch dargestellt. Auf Probleme, die meist erst durch die Programmierung erkannt werden, wird hingewiesen.

3. Leister, Müller, Stößer: *Fotorealistische Computeranimation*, Springer-Verlag, 1991 [LMS91].
   Für die Vertiefung des Strahlverfolgungsverfahrens und der Animation geeignet. Auch hier wird eine Eingabesprache – VERA – genutzt, um die Aspekte der Animation mit dem Raytracing-Verfahren zu erläutern. Das Glossar ist extrem umfangreich und liebevoll gestaltet. Die Software ist nicht mit dem Buch verfügbar.
4. Duin, Symanzik, Claussen: *Beleuchtungsverfahren in der Computergrafik*, Springer-Verlag, 1993 [DSC93].
   Für die Vertiefung von Raytracing und Radiosity gleichermaßen geeignet. Es ist eine eher wissenschaftliche Darstellung, die die gemeinsamen physikalischen und lichttechnischen Grundlagen der Beleuchtungsverfahren betont. In einem Anhang werden Implementierungshinweise gegeben.
5. Glassner (Hrsg.): *Graphics Gems*, 1990.
   Arvo (Hrsg.): *Graphics Gems II*, 1991.
   Kirk (Hrsg.): *Graphics Gems III*, 1992.
   Die Bücher sind alle bei Academic Press erschienen [Gla90,Arv91,Kir92]. „Graphics Gems" sind eine wahre Fundgrube und unerschöpfliche Ideenlieferanten. In ihnen sind Algorithmen, Datenstrukturen, Programme und Methoden der grafischen Datenverarbeitung gesammelt worden, die nirgend sonst zu finden sind, aber die fast jeder benötigt. In Form von kleinen Artikeln wird z.B. das Problem erörtert, auf einem Rasterbildschirm schnell Pixelbilder um beliebige Winkel zu drehen, schnell Quadratwurzeln zu ziehen oder eben ein Bild mit dem Radiosity-Verfahren zu berechnen. Dem letzten Band ist alternativ eine Diskette mit der Software aller drei Bände für PCs bzw. MacIntosh-Rechner beigefügt.

Neben diesen gibt es viele weitere Bücher in deutscher und englischer Sprache, die hier nicht einzeln vorgestellt oder erwähnt werden (z.B. [ES88,Gri92, Har88,Fel92]).

Als führende Fachzeitschriften sind die in der ACM SIGGRAPH erscheinende *Computer Graphics* mit den darin enthaltenen Tagungsbänden *der* Computergrafikkonferenz SIGGRAPH und die Zeitschrift *Computer Graphics & Applications* des IEEE zu nennen. Beide stellen Originalartikel vor, erstere über das gesamte Spektrum der Computergrafik, letztere in eher anwendungsorientierten Bereichen.

1.4 Sie sind gefragt ...

Es ist sicher nicht ganz einfach, mit vier Autoren an zwei Standorten ein homogen erscheinendes Buch zu schreiben. Dennoch sind wir der Meinung, daß es uns weitgehend gelungen ist. Wir freuen uns auf das Echo der Leserinnen und Leser, auf Anregungen und Kritik ebenso wie auf Lob. Haben Sie also „pfiffige"

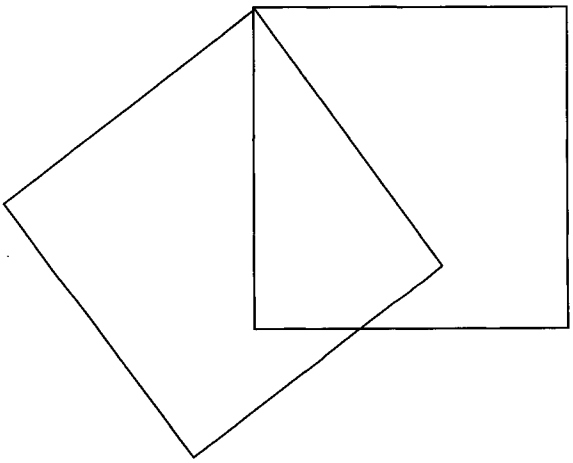
Erweiterungen der Software vorgenommen oder ein ansprechendes Bild erzeugt, wir freuen uns über einen Kontakt.

Und nun ... Viel Spaß beim Lesen!

I

2D-Algorithmen

Jürgen Plate



2. Koordinatensysteme

Die Position eines Punktes, einer Linie oder eines anderen grafischen Objektes auf der Zeichenfläche (Bildschirm, Plotterpapier, etc.) kann nur dann eindeutig festgelegt werden, wenn irgendeine Vereinbarung über die Aufteilung der Zeichenfläche besteht. Es wird in der Regel von einem kartesischen Koordinatensystem ausgegangen: die positive x-Achse ist waagerecht und rechts vom Ursprung, die positive y-Achse ist senkrecht und oberhalb des Ursprungs. Weiter ist ein Punkt in der Ebene bei diesem Koordinatensystem durch ein Koordinatenpaar $\begin{pmatrix} x & y \end{pmatrix}$ definiert. Die beiden Werte ergeben sich durch eine Projektion im rechten Winkel auf die jeweilige Koordinatenachse x bzw. y (Abb. 2.1).

Daß das nicht immer so sein muß, zeigt beispielsweise der Bildschirm des PC. Dort liegt der Ursprung in der linken oberen Ecke. Um beim PC im gewohnten System zeichnen zu können, muß man die Koordinaten auf das übliche kartesische Koordinatensystem transformieren. Diese Transformation von Koordinaten des Benutzers auf Koordinaten des Bildschirms (oder der Zeichenfläche) wird in diesem Kapitel näher behandelt.

Lösen wir zunächst das einfache Problem mit den PC-Koordinaten. Der Punkt P, der in Abb. 2.1 die Koordinaten $\begin{pmatrix} 100 & 142 \end{pmatrix}$ besitzt, soll beim PC am oberen Rand des Bildschirmes liegen. Um den Ursprung in die untere Ecke zu bringen, muß man ein wenig rechnen. Im kartesischen Koordinatensystem hat der PC-Ursprung die Koordinaten $\begin{pmatrix} 0 & y_{max} \end{pmatrix}$, wobei mit y_{max} die maxi-

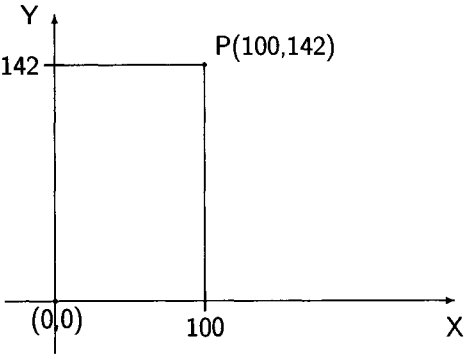


Abb. 2.1 Kartesisches Koordinatensystem

male Auflösung in y-Richtung bezeichnet wird. . Es müssen also alle Punkte $\begin{pmatrix} x & y \end{pmatrix}$ des kartesischen Koordinatensystems auf die Punkte $\begin{pmatrix} x & y_{max} - y \end{pmatrix}$ des PC-Systems abgebildet werden, um an der gewünschten Stelle zu erscheinen. Der mathematische Hintergrund für die Koordinatentransformation wird in den folgenden Abschnitten behandelt. Es wird dabei vorausgesetzt, daß Sie einige Grundlagen der Abbildungsgeometrie beherrschen (Vektoren, Matrizen, Determinanten). Es reicht im übrigen das Wissen aus, das vom Gymnasium her hängengeblieben ist.

2.1 Koordinatentransformation

Bei fast allen Zeichengeräten ist die Bildschirm- oder Zeichenfläche begrenzt. Außerdem ist das zugrundeliegende Koordinatensystem meist recht primitiv. Die linke untere Ecke (manchmal auch die linke obere Ecke) ist der Koordinatenursprung – das ist der Punkt mit den Koordinaten $\begin{pmatrix} 0 & 0 \end{pmatrix}$. Die Skalierung ist recht einfach: beim Bildschirm ist es die Zahl der möglichen Bildpunkte in x- und y- Richtung, beim Plotter meist die Zahl der Plotterschritte in x- und y-Richtung. Diese Schritte sind häufig 0.1, 0.05 oder 0.01 Millimeter groß, was zumindest eine Skalierung in Millimeter erlaubt.

Diese Skalierung entspricht aber fast nie den Wünschen des Benutzers. Will man seine Zeichnung in beliebigen Größen skalieren, muß man die tatsächlichen Koordinaten auf die Bildschirmgegebenheiten abbilden – per Koordinatentransformation.

2.2 Weltkoordinaten und Bildkoordinaten

Das Koordinatensystem soll an die Größenordnungen und Anforderungen des speziellen Problems angepaßt sein (Meter, Millimeter, Grad, Sekunden, Millionen Einwohner, Jahre, etc.). Dieses Weltkoordinatensystem muß auf die Koordinaten des Gerätes abgebildet werden. Ein rechteckiges Fenster des Weltkoordinatensystems (Window) wird auf ein beliebiges Fenster der zur Verfügung stehenden Zeichenfläche abgebildet (Viewport). Abb. 2.2 zeigt diesen Sachverhalt. Im deutschen Sprachgebrauch gibt es manchmal Probleme bei der Zuordnung der amerikanischen Begriffe „Window“ und „Viewport“ . Daher hier die Zuordnung noch einmal:

„Window“ :	Weltkoordinaten, Skalierung
„Viewport“ :	Gerätekoordinaten, Fenster

2.3 Skalierung und Fenster

Es muß also ein beliebiger Punkt im Weltkoordinatensystem $P = \begin{pmatrix} x & y \end{pmatrix}$ in den entsprechenden Punkt $P' = \begin{pmatrix} fx & fy \end{pmatrix}$ des Fensters im Gerätekoor-

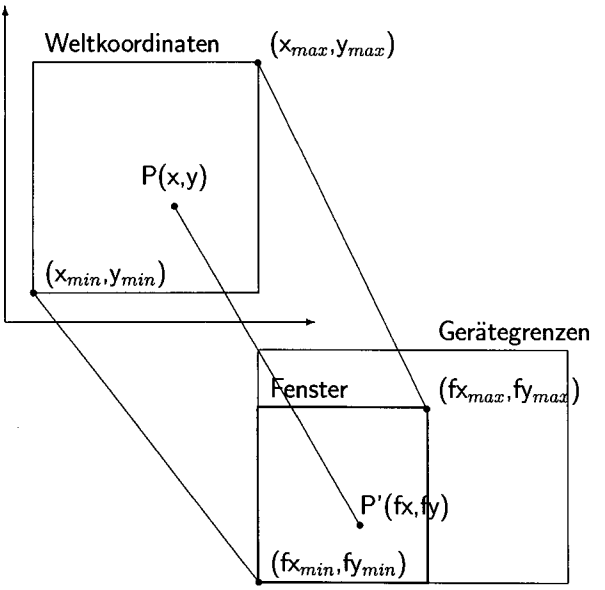


Abb. 2.2 „Window" und „Viewport"

dinatensystem umgerechnet werden. Im allgemeinen werden zwei rechteckige Bereiche aufeinander abgebildet. Dazu muß zunächst einiges definiert werden:

Die Zeichenfläche hat eine maximale Breite b und eine maximale Höhe h. Auf der Zeichenfläche wird ein Fenster definiert, das durch die Werte A, B, C und D begrenzt wird (Abb. 2.3). Das Fenster $ABCD$ wird nun von $x_{min} - x_{max}$ (Strecke zwischen A und B) und von $y_{min} - y_{max}$ (Strecke zwischen C und D) skaliert. Es gilt:

$$\begin{pmatrix} x_{min} & y_{min} \end{pmatrix} \longrightarrow \begin{pmatrix} A & C \end{pmatrix} = \begin{pmatrix} fx_{min} & fy_{min} \end{pmatrix}$$
$$\begin{pmatrix} x_{max} & y_{max} \end{pmatrix} \longrightarrow \begin{pmatrix} B & D \end{pmatrix} = \begin{pmatrix} fx_{max} & fy_{max} \end{pmatrix}.$$

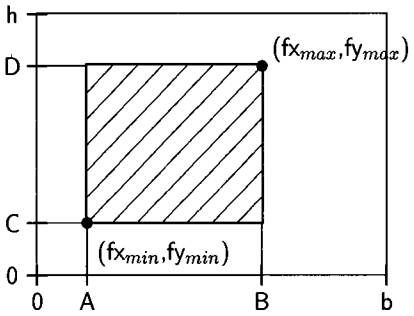


Abb. 2.3 Zeichenfläche mit Fenster

Nehmen wir zunächst an, der linke untere Eckpunkt des Fensters würde sich mit dem Ursprung decken. Dann gilt generell:

$$fx = P \cdot x \qquad \text{und} \qquad fy = Q \cdot y$$

mit:

$$P = \frac{(B - A)}{(x_{max} - x_{min})} \qquad \text{und} \qquad Q = \frac{(D - C)}{(y_{max} - y_{min})}.$$

Die Strecken werden also linear aufeinander abgebildet. Liegt das Fenster an beliebiger Stelle auf dem Bildschirm, kommt noch eine Translation der Ursprungskoordinaten hinzu. Die Formel lautet dann:

$$fx = P \cdot x + S \qquad \text{und} \qquad fy = Q \cdot y + T.$$

Für S und T gilt:

$$\begin{aligned} S &= \frac{(A \cdot x_{max} - B \cdot x_{min})}{x_{max} - x_{min}} \\ T &= \frac{(C \cdot y_{max} - D \cdot y_{min})}{y_{max} - y_{min}}. \end{aligned}$$

2.4 Transformationen

In diesem Abschnitt ist ein kleiner Ausflug in die analytische Geometrie angebracht: Dabei wollen wir nicht sämtliche theoretischen Grundlagen aufrollen, sondern nur einige Sachverhalte aus der Schulgeometrie ins Gedächtnis rufen. Der Punkt mit der Koordinate $\begin{pmatrix} x & y \end{pmatrix}$ wird durch den Vektor

$$\begin{pmatrix} x \\ y \end{pmatrix}$$

in unserem kartesischen Koordinatensystem eindeutig festgelegt.
Lineare Abbildungen lassen sich durch Matrizen darstellen:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}.$$

Es sind affine Abbildungen, das heißt:

- sie sind bijektiv (umkehrbar eindeutig),
- sie sind geradentreu und
- das Teilverhältnis von Strecken ist invariant unter der Abbildung.

Das führt zu den folgenden Eigenschaften:

- Ein nichtentartetes Dreieck wird auf ein beliebiges nicht entartetes Dreieck abgebildet.
- Parallele Geraden werden auf parallele Geraden abgebildet.

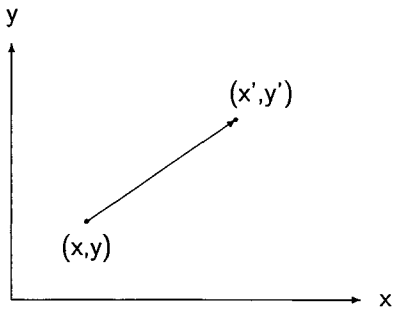


Abb. 2.4 Translation

- Das Flächenverhältnis von Original und Bild ist gleich der Determinanten der Abbildungsmatrix.

Die Determinante der Abbildungsmatrix muß daher ungleich Null sein:

$$a_{11} \cdot a_{22} - a_{12} \cdot a_{21} \neq 0$$

Die 2D-Abbildung hat die allgemeine Form:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}.$$

Ähnlichkeitsabbildungen sind affine Abbildungen mit der Eigenschaft, daß das Verhältnis von Bildstrecke zur Originalstrecke eine Konstante ist, die man „Ähnlichkeitsverhältnis" nennt.

Sie haben die allgemeine Form:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} k\cos\beta & \pm k\sin\beta \\ k\sin\beta & \pm k\cos\beta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}.$$

Hat das Ähnlichkeitsverhältnis den Betrag 1, erhalten wir die Kongruenzabbildung, deren allgemeine Form lautet:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}.$$

Hier wollen wir einige Sonderformen näher unter die Lupe nehmen. Da wäre als einfache Abbildung die Translation (Verschiebung), bei der $\beta = 0$ ist (Abb. 2.4). Wenn das Bildfenster auf der Zeichenfläche verschoben wird, dann werden alle Koordinaten einer Translation unterzogen.

Die nächste Abbildung ist die Drehung um einen beliebigen Punkt $\begin{pmatrix} m_1 & m_2 \end{pmatrix}$, die in Abb. 2.5 zu sehen ist. Als letztes brauchen wir noch eine Achsenspiegelung mit der Spiegelachse durch den Ursprung (Abb. 2.6). Die Spiegelachse hat die Steigung $\tan(\beta/2)$.

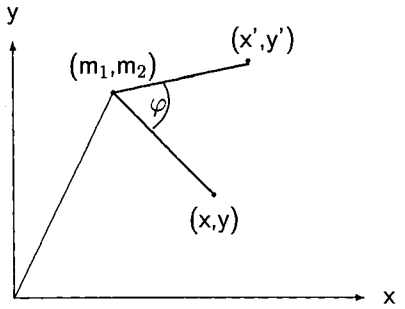


Abb. 2.5 Drehung

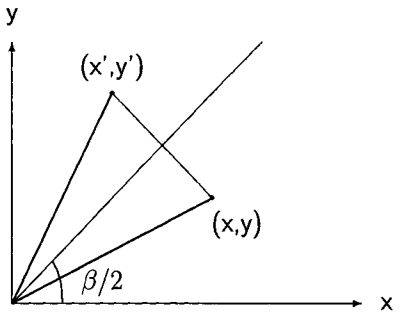


Abb. 2.6 Spiegelung

Mit den genannten Abbildungen können alle notwendigen Koordinatentransformationen vorgenommen werden. Nehmen wir als einfachstes Beispiel das Problem mit den PC-Koordinaten von weiter vorne. Sie erinnern sich sicher: der Nullpunkt saß in der oberen statt in der unteren linken Ecke. Wir hatten dort gesagt:

$$x' = x \qquad \text{und} \qquad y' = y_{max} - y.$$

Es lag also eine Spiegelung an der Geraden parallel zur x-Achse mit $y = y_{max}/2$ vor. Abbildungsgeometrisch war das die Kombination einer Spiegelung und einer Translation, also:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}.$$

„Parallel zur x-Achse" bedeutet $\beta = 0$. Außerdem ist $t_1 = 0$ und $t_2 = y_{max}$, also:

$$\begin{aligned} x' &= x \\ y' &= -y + y_{max} = y_{max} - y. \end{aligned}$$

Auch unsere Abbildung der Weltkoordinaten auf die Bildschirmkoordinaten läßt sich mit Abbildungsmatrizen darstellen. Mit Hilfe der Abbildungsgeometrie lassen sich also beliebige Figuren auf der Zeichenfläche frei verschieben, drehen, vergrößern und verkleinern. Dazu noch ein letztes Beispiel:

Schriftzeichen werden beim Plotter als Folge von Vektoren definiert. Es wird für jedes Zeichen eine Liste von Koordinaten vorgegeben, die nacheinander vom Stift angefahren werden. Durch eine Translation ist es möglich, an beliebiger Stelle einen Buchstaben zu zeichnen. Nun wäre es wünschenswert, die Buchstaben in beliebiger Größe zeichnen zu können oder sie zu drehen. Sehen wir uns dazu einmal die verschiedenen Abbildungsmöglichkeiten an:

1. Drehung um den Winkel β:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\beta & -\sin\beta \\ \sin\beta & \cos\beta \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

2. Einstellen auf Breite b , Höhe h:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} b & 0 \\ 0 & h \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

3. Schrägstellen, Kursivschrift:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 1 & 0.25 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Durch Kombination dieser Abbildungen kann jeder beliebige Schriftgrad eingestellt werden, zum Beispiel Breite 10, Höhe 15, Drehwinkel 45°, ohne Kursivschrift:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} b & 0 \\ 0 & h \end{pmatrix} \cdot \begin{pmatrix} \cos(45) & -\sin(45) \\ \sin(45) & \cos(45) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}.$$

Setzt man die Werte ein und multipliziert die Matrizen aus, ergibt sich:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} 7.071 & -7.071 \\ 10.607 & 10.607 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}.$$

Mit den drei Abbildungen Translation, Drehung und Skalierung kann man fast immer alle notwendigen Koordinatentransformationen vornehmen. Ich will sie zum Schluß dieses Abschnitts noch einmal als Formeln zusammenfassen:

Translation:

$$\begin{aligned} x' &= x + t_1 \\ y' &= y + t_2 \end{aligned}$$

Drehung:

$$\begin{aligned} x' &= x \cdot \cos(\beta) - y \cdot \sin(\beta) \\ y' &= x \cdot \sin(\beta) + y \cdot \cos(\beta) \end{aligned}$$

Skalierung:

$$\begin{aligned} x' &= x \cdot skal.x \\ y' &= y \cdot skal.y \end{aligned}$$

2.5 Polarkoordinaten

Geschlossene Kurven lassen sich in kartesischen Koordinaten oft nur sehr schwer darstellen (z.B. Spiralen, Kardiode, etc.). So hat z.B. die Kardiode in Polarkoordinaten die Gleichung

$$r = k(1 + \cos(\beta)),$$

in kartesischen Koordinaten die Form

$$(x^2 + y^2)^2 - 2k(x^2 + y^2) = k^2 y^2.$$

Was sind nun Polarkoordinaten und wie unterscheiden sie sich von den kartesischen (rechtwinkligen) Koordinaten?

Bei Polarkoordinaten wird jeder Punkt der Zeichenebene durch zwei Werte r und β bestimmt (Abb. 2.7). r ist der Abstand des Punktes vom Ursprung und β stellt den Winkel zwischen x-Achse und der Verbindungsgeraden vom Ursprung zum Punkt dar. Meist kommt man nur mit der Parameterform der Polarkoordinaten in Berührung:

$$x = f(t) \qquad \text{und} \qquad y = g(t).$$

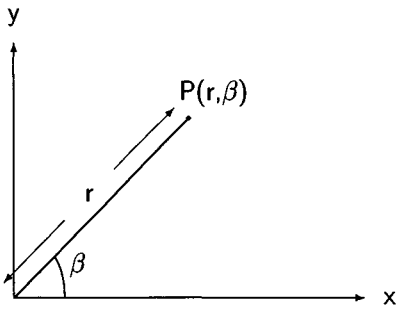


Abb. 2.7 Polarkoordinaten

Dazu ein Beispiel: Der Kreis, sicherlich eine geschlossene Kurve, hat die Gleichung:

$$x^2 + y^2 = r^2.$$

In der Parameterform wird daraus:

$$\left.\begin{array}{l} x = r \cdot \cos(\beta) \\ y = r \cdot \sin(\beta) \end{array}\right\} \qquad 0 \leq \beta \leq 2\pi$$

Wir haben bei der Kreisberechnung eine Tatsache schon ausgenützt; Polarkoordinaten und kartesische Koordinaten lassen sich ineinander umrechnen.

Die Umrechnung der Koordinatensysteme geschieht folgendermaßen:

Kartesisch in Polar	Polar in Kartesisch
$r = \sqrt{x^2 + y^2}$	$x = r \cdot \cos(\beta)$
$\beta = \arctan(y/x)$	$y = r \cdot \sin(\beta)$

2.6 Homogene Koordinaten

Die bisher behandelten Abbildungen haben noch einen Nachteil, sie besitzen immer einen Fixpunkt, und zwar den Ursprung. Es gilt für beliebige Koeffizienten:

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Will man mit Matrizen auch eine Translation (oder andere Transformationen, bei denen sich alle Punkte ändern) durchführen, muß man erreichen, daß der Fixpunkt außerhalb des Ursprungs liegt. Dazu führt man eine dritte Dimension ein und nimmt als Bildfläche die Ebene $z = 1$. Der Fixpunkt $\begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$ liegt außerhalb der Bildebene. Die Transformationen werden nun durch 3×3-Matrizen beschrieben, deren untere Zeile im allgemeinen aus dem

Vektor $\begin{pmatrix} 0 & 0 & 1 \end{pmatrix}$ besteht. Alle Koordinatenpunkte erhalten dann eine dritte Komponente, die immer 1 ist und beim Ergebnis dann weggelassen werden kann:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Mit den so gewonnenen homogenen Koordinaten wird genau so gerechnet wie mit den einfachen Matrizen.

Gehen wir die Abbildungstypen von Abschnitt 2.4 noch einmal durch:

1. Skalieren um die Höhe h und Breite b:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} b & 0 & 0 \\ 0 & h & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} b \cdot x \\ h \cdot y \\ 1 \end{pmatrix}$$

2. Drehung um den Winkel β:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos(\beta) - y\sin(\beta) \\ x\sin(\beta) + y\cos(\beta) \\ 1 \end{pmatrix}$$

3. Spiegelung an einer Geraden, die den Winkel β mit der x-Achse einschließt. Hier wird die Abbildung in 3 Stufen ermittelt:
   (a) Drehung um $-\beta$, d.h. Abbildung auf die Koordinatenachsen
   (b) Spiegelung an der x-Achse
   (c) Drehung um $+\beta$, um den ersten Schritt wieder rückgängig zu machen

   Die Spiegelung an der x-Achse hat die Matrix:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

   Versuchen Sie doch einmal, die Matrix für eine Spiegelung an der y-Achse aufzustellen.

   Die gesamte Abbildung sieht dann so aus:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(-\beta) & -\sin(-\beta) & 0 \\ \sin(-\beta) & \cos(-\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Das ergibt nach Ausmultiplizieren und Umformen der Winkelfunktionen die folgende Abbildung:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos(2\beta) & \sin(2\beta) & 0 \\ \sin(2\beta) & -\cos(2\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}.$$

Wenn Sie sich erinnern, dann war die Steigung der Spiegelgeraden ja auch genau $\tan(\beta/2)$.

4. Translation um U in x-Richtung und V in y-Richtung:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & U \\ 0 & 1 & V \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x+U \\ y+V \\ 1 \end{pmatrix}.$$

Damit wären die für uns wichtigsten Abbildungstypen abgehandelt. Durch Hintereinanderausführen dieser Grundabbildungen lassen sich beliebige Transformationen erreichen. Was dabei zu beachten ist, steht im nächsten Abschnitt.

2.7 Zusammensetzen von Transformationen

Wie Sie bei der Spiegelung schon gesehen haben, ist es manchmal notwendig, Abbildungen hintereinander zu schalten, um das gewünschte Ziel zu erreichen. Da es sehr rechenaufwendig ist, die angegebenen Abbildungsmatrizen für jeden zu berechnenden Punkt wirklich hintereinander auszuführen, konstruiert man vielmehr eine Gesamttransformation aus den einzelnen Abbildungen und unterwirft die Punkte dann dieser einen Transformationsmatrix.

Dazu ein Beispiel: Das Koordinatensystem soll um den Punkt $\begin{pmatrix} m_1 & m_2 \end{pmatrix}$ gedreht werden, und zwar um den Winkel β. Die Abbildung wird in drei Schritten durchgeführt. Zuerst wird der Punkt $\begin{pmatrix} m_1 & m_2 \end{pmatrix}$ zum Ursprung gemacht (Translation um $\begin{pmatrix} -m_1 & -m_2 \end{pmatrix}$). Dann wird um den Ursprung gedreht und zum Schluß der alte Ursprung wieder hergestellt (Translation um $\begin{pmatrix} m_1 & m_2 \end{pmatrix}$). Also ergibt sich eine Gesamtabbildung von:

$$\begin{aligned} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} &= \begin{pmatrix} 1 & 0 & m_1 \\ 0 & 1 & m_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \cos(\beta) & -\sin(\beta) & 0 \\ \sin(\beta) & \cos(\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \\ &\quad \cdot \begin{pmatrix} 1 & 0 & -m_1 \\ 0 & 1 & -m_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos(\beta) & -\sin(\beta) & m_1 \\ \sin(\beta) & \cos(\beta) & m_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & -m_1 \\ 0 & 1 & -m_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} \cos(\beta) & -\sin(\beta) & -m_1\cos(\beta) + m_2\sin(\beta) + m_1 \\ \sin(\beta) & \cos(\beta) & -m_1\sin(\beta) - m_2\cos(\beta) + m_2 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}. \end{aligned}$$

Zweckmäßigerweise berechnet man zuerst die Werte von $\cos(\beta)$ und $\sin(\beta)$, dann die Werte in der dritten Spalte und setzt diese Werte in die Matrix ein. Mit dieser Matrix, die dann nur noch Zahlenwerte enthält, werden dann die neuen Bildpunkte ermittelt.

Erweitert man die homogenen Koordinaten um eine weitere Dimension, können dreidimensionale Körper bearbeitet werden.

3. Algorithmen

3.1 Abschneiden (Clipping) und Überdecken

Unter Clipping versteht man das Abschneiden der Teile eines Bildes, die außerhalb eines vorher vereinbarten Fensters liegen. Bei manchen Computern ist das sehr wichtig, da sie das Programm beim Versuch, außerhalb der verfügbaren Bildfläche zu zeichnen, mit einem Fehler abbrechen. Bei modernen Plottern ist der Clipping-Algorithmus als Randüberwachung eingebaut. Will man Fenster auf der Zeichenfläche einführen, braucht man ihn auf jeden Fall. Dabei ist es gar nicht so einfach, die Grafik-Software zu veranlassen, eine Linie, die zu einem Punkt außerhalb des Fensters führt, richtig abzuschneiden.

Ein Clipping-Algorithmus ist für die Grundsoftware von Grafik-Software-Paketen und für intelligente Grafikausgabegeräte notwendig. Er muß also sicher und schnell arbeiten. Wir können von der Annahme ausgehen, daß das Fenster immer ein achsenparalleles Rechteck ist, für das gilt:

$$x_{min} \leq x \leq x_{max} \qquad \text{und} \qquad y_{min} \leq y \leq y_{max}.$$

Um Rechenzeit zu sparen, sollte man schon im Weltkoordinatensystem abschneiden, so daß außerhalb des Fensters liegende Teile der Zeichnung auch nicht mehr irgendwelchen Abbildungen unterworfen werden. Ganz einfach wäre natürlich das Clipping, wenn man bei einem Grafiktreiber für die Bildschirmgrafik eine Primitivoperation zum Setzen eines einzelnen Bildpunktes hat, auf der alle weiteren Operationen zum Zeichnen von Linien und Kreisen aufbauen. Die Routine könnte dann so aussehen:

```
PROCEDURE Point(x, y : INTEGER);
(* Die Fenstergrenzen Xmin, Xmax, Ymin, Ymax sind global *)
BEGIN
  IF (Xmin<=x) AND (x<=Xmax) AND (Ymin<=y) AND (y<=Ymax) THEN
     (* setze Bildpunkt (x,y) *)
  ELSE
     (* tue nichts *)
END;
```

Da letztlich alle Zeichnungen aus (mitunter recht kurzen) geraden Linien bestehen, kann man sich auf das Clippen von Linien beschränken. Ein entsprechender Algorithmus wurde von Cohen und Sutherland entwickelt. Er geht davon aus, daß jede Linie höchstens einen einzigen sichtbaren Teil haben kann (Abb. 3.1).

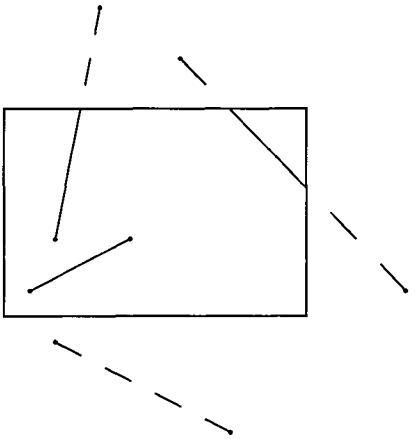


Abb. 3.1 Mögliche Anordnungen einer Linie

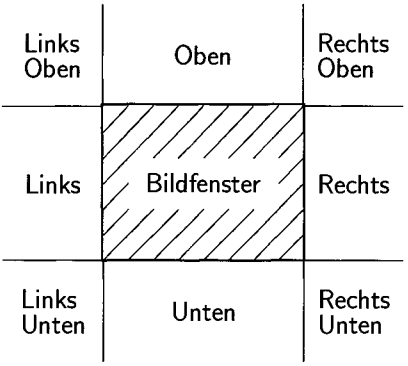


Abb. 3.2 Einteilung des Raums um das Fenster

Verlängert man die Kanten des Fensters zu Geraden, wird die Ebene in neun Teile zerlegt. Man kann durch einfache Vergleiche feststellen, in welcher Region außerhalb des Fensters ein Punkt liegt (Abb. 3.2).

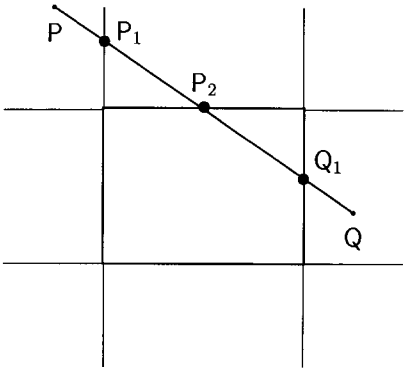


Abb. 3.3 Abschneiden einer Linie

Der Clipping-Algorithmus von Cohen und Sutherland lautet nun:

Algorithmus 3.1 (Clipping nach Cohen und Sutherland)

Schritt 1: *Man ordnet jedem Punkt die Regionen, in denen er liegt, mengentheoretisch zu. Elemente der Menge sind die Eigenschaften „links", „rechts", „oben" und „unten". Liegt ein Punkt rechts unterhalb des Fensters, wird ihm die Menge „rechts, unten" zugeordnet.*

Schritt 2: *Bilden die beiden Endpunkte einer Linie die leere Menge, ist ihre Vereinigung also leer, ist die Linie vollständig sichtbar; man ist fertig.*

Schritt 3: *Ist der Durchschnitt beider Mengen nicht leer, dann ist die gesamte Linie außerhalb des Fensters und man ist auch fertig.*

Schritt 4: *In allen übrigen Fällen nimmt man den Eckpunkt, dem eine nichtleere Menge zugeordnet ist. Dieser wird ersetzt durch den Schnittpunkt mit der entsprechenden Begrenzungsgeraden des Fensters. Mit der so gekürzten Linie wird dann bei Schritt 2 fortgefahren. Die Linie $\overline{PQ}$ in Abb. 3.3 wird also zunächst auf $\overline{P_1Q}$, dann auf $\overline{P_2Q}$ und schließlich $\overline{P_2Q_1}$ reduziert. Gezeichnet wird dann $\overline{P_2Q_1}$.*

Überdecken ist das exakte Gegenteil vom Abschneiden. Nur diesmal müssen die Linien innerhalb des Rechteckfensters entfernt werden. Man kann also mit dem gleichen Algorithmus arbeiten, wie beim Abschneiden. Die Linie $\overline{PQ}$ in Abb. 3.3 würde also zu zwei Linien werden: $\overline{PP_2}$ und $\overline{Q_1Q}$.

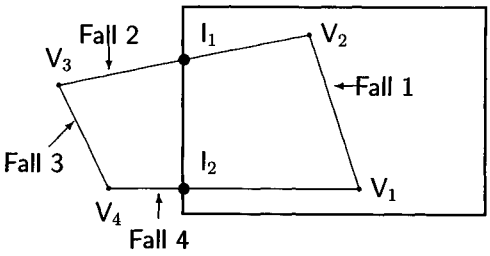


Abb. 3.4 Polygon-Clipping

3.2 Polygon-Clipping

Beim Clippen von Polygonen kann der obige Algorithmus nicht verwendet werden, da der geschlossene Linienzug unterbrochen würde. Ein von Sutherland und Hodgeman entwickelter Algorithmus bringt hier Abhilfe. Es wird ein Polygon in Form einer Reihe von miteinander zu verbindenden $\begin{pmatrix} x & y \end{pmatrix}$-Koordinaten übergeben, wobei implizit angenommen wird, daß der letzte Punkt wieder mit dem ersten verbunden wird. Beim Clipping werden die Linien an den Fenstergrenzen abgeschnitten und durch Kanten entlang der Fenstergrenzen ersetzt. Die Clipping-Routine ersetzt also die Koordinatentabelle durch eine geänderte Tabelle, die den Linienzug innerhalb des Fensters enthält. Betrachten Sie zur Erläuterung die Abb. 3.4.

Das Polygon hat die Eckpunkte V_1, V_2, V_3 und V_4. Dabei lassen sich vier verschiedene Fälle unterscheiden:

1.Fall: Die Line $\overline{V_1V_2}$ liegt voll innerhalb des Fensters, somit wird V_2 in die neue Liste unverändert übernommen.

2.Fall: Die Line $\overline{V_2V_3}$ schneidet eine Kante des Fensters. Es wird der Schnittpunkt I_1 berechnet und in die Liste übernommen.

3.Fall: Die Linie $\overline{V_3V_4}$ liegt vollständig außerhalb des Fensters, es wird kein Punkt in die neue Liste übernommen.

4.Fall: Die Linie $\overline{V_4V_1}$ schneidet die Fensterkante, es wird der Schnittpunkt I_2 berechnet und in die Liste übernommen.

Aus dem Polygon V_1, V_2, V_3, V_4 wird so das Polygon V_2, I_1, I_2, V_1. Damit auch Schnitte berechnet werden, die „über Eck" liegen, muß das Clipping für jede Kante des Fensters (also viermal) durchgeführt werden – das Abschneiden und Ergänzen der Kanten erfolgt dann schrittweise.

3.3 Flächenfüllen und Schraffieren

Ein Algorithmus zum (farbigen) Füllen von Flächen wird in der Präsentationsgrafik bei der Zeichnung von Werkstücken mit Schnittflächen und im CAD-Bereich benötigt. Man muß bei den Verfahren zum Füllen von Flächen zwischen Routinen für den Bildschirm und solchen für den Plotter unterscheiden.

Die Möglichkeit, beim Bildschirm die Farbe eines Bildpunktes feststellen zu können, macht die Füllalgorithmen sehr viel einfacher. Da das Füllen immer als Folge von Punkten, also in Form von Linien erfolgt, kann man das Schraffieren als Sonderform des Flächenfüllens betrachten oder das Füllen als Sonderform des Schraffierens.

Da die Schrittweiten beim Plotter oder Bildschirm immer eine minimale, ganzzahlige Länge besitzen, kann man jede darstellbare Fläche, also auch eine Kreisfläche, als Polygon (= Vieleck) betrachten, also als eine, von beliebigen geraden Linien begrenzte Fläche. Sie erinnern sich: auch der Plotter setzt Linien aus Einzelschritten zusammen. Diese Polygone sind in fast allen Fällen durch die Folge ihrer Eckpunkte definiert.

3.3.1 Flächenfüllen am Bildschirm

Mit Fläche ist hier eine Region gemeint, die durch eine ununterbrochene Reihe von Bildpunkten umschlossen ist. Nebenbei – alles, was hier über Schwarzweißgrafik gesagt wird, läßt sich auch auf Farbe übertragen. Unsere Aufgabe ist es, ein weißes Polygon mit dunkler Innenfläche weiß auszufüllen. In nahezu allen Realisierungen von Grafikbildschirmen läßt sich feststellen, ob ein bestimmter Bildpunkt (=Pixel) hell oder dunkel ist. Dann ergibt sich als erster Ansatz für einen Flächenfüllalgorithmus in schwarz-weiß:

1. Wähle einen Punkt innerhalb des Polygons.
2. Ist der Punkt schwarz, dann färbe ihn weiß.
3. Versuche es mit allen vier benachbarten Punkten.
4. Sind alle Nachbarpunkte weiß, dann ist das Polygon gefüllt.

Es bietet sich also das folgende rekursive Programmstück an.

```
PROCEDURE Fuelle (X, Y : INTEGER);
BEGIN
  IF Pixel(X,Y) = schwarz THEN BEGIN
    Pixel(X,Y) := weiss;
    Fuelle(X,Y - 1);
    Fuelle(X,Y + 1);
    Fuelle(X - 1,Y);
    Fuelle(X + 1,Y);
  END;
END;
```

Der Nachteil der Prozedur liegt auf der Hand. Durch die rekursiven Aufrufe wird beliebig viel Speicher belegt und der Algorithmus ist zudem recht langsam. Es muß also eine Lösung gefunden werden, die nicht rekursiv ist und effizienter arbeitet. Um das zu erreichen, bearbeitet man die Fläche nicht Punkt für Punkt, sondern zeilenweise. Betrachten Sie dazu Abb. 3.5. Der Algorithmus ist komplexer.

Algorithmus 3.2 (Flächenfüllen)

1. *Beginne etwa in der Bildmitte.*
2. *Fülle diejenige Zeile mit weißen Punkten, in der der Startpunkt liegt.*
3. *Untersuche die darüberliegende Zeile von rechts nach links, und speichere alle Punkte, deren Nachbar bereits hell ist, in dieser Reihenfolge auf einem Stack.*
4. *Verfahre genauso mit allen Punkten in der darunterliegenden Zeile.*
5. *Verwende das oberste Element des Stacks als neuen Startpunkt. Wenn der Stack nicht leer ist, mache weiter bei Schritt 1, im anderen Fall ist das Polygon gefüllt.*

Damit der Algorithmus nicht ins Leere läuft, wenn der Bildrand erreicht ist, kann man zusätzlich eine Randüberwachung einbauen, die den Wert „weiß" zurückgibt, wenn der Rand erreicht ist.

Diese Prozedur läßt sich recht einfach auf das Füllen mit Mustern erweitern. Der Bildpunkt wird nicht einfach eingefärbt, sondern aus einem vordefinierten Muster genommen.

3.3.2 Schraffieren

Hier geht es nicht ganz so leicht, denn es sind meist nur die Eckpunkte des Rechtecks bekannt, nicht aber die Punkte der Verbindungslinien. Außerdem ist das Berechnen der Linienpunkte viel zu aufwendig.

Die Programmteile für waagerechte und vertikale Linien sind einfach. Bei den schrägen Linien stellt sich das Problem, daß in der unteren und oberen Ecke die Linien richtig abgeschnitten werden müssen. Wir können an dieser Stelle auch den generellen Clipping-Algorithmus verwenden.

Das Problem bei der Schraffur von Polygonflächen ist das Hinterschneiden von Kanten. Eine Schraffurlinie besteht unter Umständen aus mehreren Teilstücken. Da in der Regel auch nur die Koordinaten der Eckpunkte des Polygonzugs bekannt sind, müssen für jede Schraffur gerade die Schnittpunkte mit den Kanten (Verbindunglinien) des Polygons berechnet werden.

Ein Punkt auf der Schraffurgeraden befindet sich genau dann innerhalb des Polygons, wenn ab diesem Punkt die Zahl der Schnittpunkte zwischen der Schraffurgeraden und dem Polygon ungerade ist. Mit anderen Worten muß der erste Schnittpunkt mit dem zweiten, der dritte mit dem vierten verbunden werden und so weiter. Gefährlich wird es, wenn eine Schraffurlinie auf einen Eckpunkt des Polygons trifft.

In der Literatur sind zwei Algorithmen für die Polygonschraffur weit verbreitet, die auch noch eine annähernd gleiche Bezeichnung tragen: der X-Y-Algorithmus und der Y-X-Algorithmus. Zum zweiten gehört der Plotter-Algorithmus [Pla87]. Dieser arbeitet nach dem Y-X-Algorithmus und hat einen geringeren Rechenaufwand. Der Algorithmus benutzt Plotterkoordinaten und läßt Schraffurlinien mit beliebigem Winkel und Abstand zu.

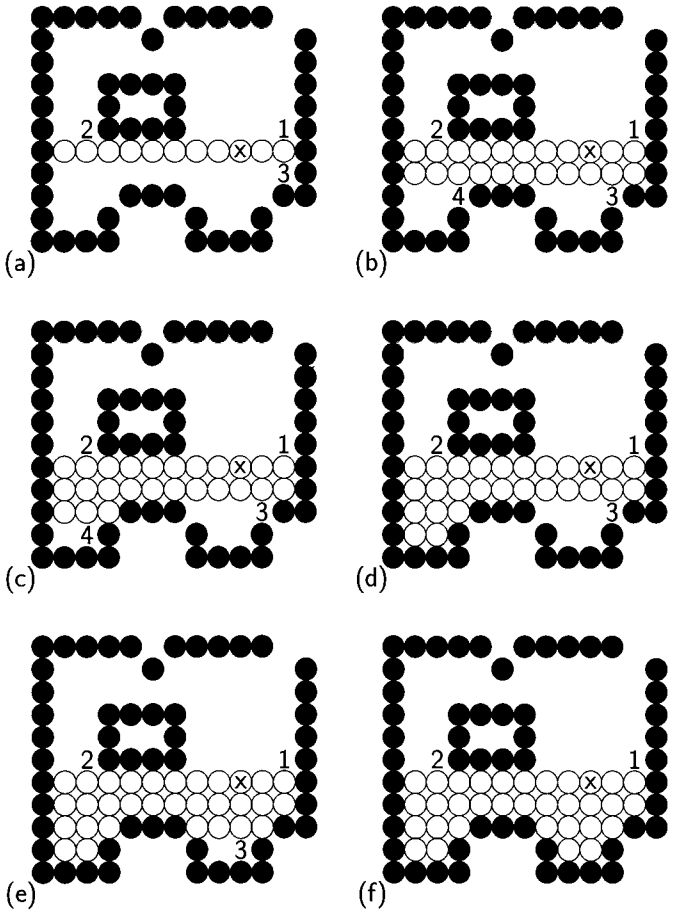


Abb. 3.5 Flächenfüllen auf dem Rasterschirm. Der Startpunkt des Algorithmus ist mit einem x gekennzeichnet. Zur Verdeutlichung ist der Rand des Polygons mit schwarzen Kreisen, die durch den Algorithmus gesetzten Pixel mit leeren Kreisen dargestellt.

4. Anwendungen der 2D-Grafik

Im folgenden werden häufige Anwendungsfälle der 2D-Grafik übersichtsartig zusammengefaßt. Ein Teil davon wird wegen ihrer Wichtigkeit für die dreidimensionale Modellierung gründlicher besprochen. Programme für diese Anwendungen wurden bereits früher veröffentlicht [Pla87].

4.1 Darstellen von Funktionen und Wertetabellen

Häufig kommt es vor, daß sich die abzubildenden Kurven in Form einer Funktion $y = f(x)$ darstellen lassen oder daß es sich um eine Reihe von Meßwerten handelt, die über der x-Achse aufgetragen werden.

Der Graph einer *stetigen Funktion* ist eine Kurve, die „in einem Zug ohne Abheben des Zeichenstiftes" durchgezogen werden kann. Dabei sind Knickstellen erlaubt, Sprungstellen dagegen nicht. Ein Programm, das bei Angabe der Intervallgrenzen für x die Funktion $y = f(x)$ zeichnet und dabei die Intervallgrenzen für y automatisch bestimmt, berechnet in einem ersten Schritt die y-Werte und speichert sie in einem Feld. Gleichzeitig werden der größte und der kleinste Wert im angegebenen Intervall ermittelt. Mit diesen Werten y_{min} und y_{max} kann man die Zeichnung skalieren. Das Werteintervall von y_{min} bis y_{max} muß auf den darstellbaren Bereich abgebildet werden.

Bei der x-Achse geht man analog vor.

Hat die zu zeichnende Funktion *Sprungstellen*, läßt sie sich nicht so behandeln. Auch beim Zeichnen von Wurzelfunktionen oder logarithmischen Funktionen gibt es Probleme, sobald x kleiner als Null wird.

Die Behandlung von Unstetigkeitsstellen geschieht durch interaktive Festlegung des Bereiches für die y-Werte. So kann das Intervall auf interessante Bereiche festgelegt werden.

Weiter vorne war zu lesen, daß sich etliche Kurven in *Parameterform* viel besser beschreiben lassen. Die Gleichungen lauten:

$$x = f(t) \qquad \text{und} \qquad y = g(t).$$

Solche Kurven lassen sich durch Variation des Parameters t gut zeichnen. Beispiele für diese Kurven sind z.B. die Archimedische Spirale mit der Gleichung:

$$x = Z \cdot t \cdot \cos(t) \qquad \text{und} \qquad y = Z \cdot t \cdot \sin(t).$$

und die Hypozykloide mit der Gleichung :

$$x = (R_1 - R_2)\cos(t) + L \cdot R_2 \cdot \cos(\frac{R_1 - R_2}{R_2} \cdot t)$$
$$y = (R_1 - R_2)\sin(t) + L \cdot R_2 \cdot \sin(\frac{R_1 - R_2}{R_2} \cdot t).$$

Oft liegen die Rohdaten nicht in Form einer Funktion vor, sondern als Ergebnisse von Messungen oder einer anderen Art der Datenerfassung. Wir haben also eine Reihe von *Wertepaaren* $\begin{pmatrix} x & y \end{pmatrix}$, wobei die x-Werte eine aufsteigende Folge bilden.

Nach Einführen zweier Felder für die x- und y-Werte, einer Änderung der Zeichenroutine, und durch Hinzufügen einer Eingaberoutine können wir auch solche Zahlen verarbeiten. Nicht immer eignet sich die gradlinige Verbindung von Meßpunkten für eine übersichtliche Darstellung. Sehr viel besser sind dann *Histogramme.* Um solche zu zeichnen, muß man sich nur überlegen, wie man anstelle eines Punktes ein Rechteck erhält. Dieses reicht bis zum y-Wert und steht mit seiner Mitte auf der entsprechenden x-Koordinate. Da nach wie vor auch negative Werte zugelassen sind, kann man auch Abweichungs-Diagramme zeichnen. Ein Tip noch: Wenn die x-Werte in gleichen Abständen liegen, wird die Grafik besonders ansprechend.

Säulengrafik ist eine dreidimensional wirkende Art der Präsentationsgrafik und dem Histogramm recht ähnlich. Anstelle eines einfachen Rechtecks wird diese um eine Deckfläche und eine schraffierte Seitenfläche erweitert.

Die letzte Form der Präsentationsgrafik, die ich Ihnen vorstellen will, ist die *Tortengrafik.* Hier werden prozentuale Verhältnisse als Kreissegmente dargestellt. Wichtig ist, daß die Daten als Prozentwerte eingegeben werden müssen, sonst wird die Torte ruiniert.

Oft besteht der Wunsch, nicht Prozentzahlen, sondern Rohwerte einzugeben. Dazu muß man die Rohwerte aufsummieren und in einen prozentualen Anteil umrechnen:

$$y_i = \frac{y_i \cdot 100}{\text{Summe}} \qquad i = 0, \ldots, n.$$

4.2 Lineare Approximation

Bei der Darstellung von Meßwerten, die als Folge diskreter Koordinaten vorliegen, weiß man oft über den funktionellen Zusammenhang zwischen x- und y-Werten Bescheid. Man kennt also die Form der Kurve, beispielsweise eine Gerade, weiß aber nichts über die Lage dieser Kurve. Die simpelste Lösung ist die Verbindung der Meßpunkte durch einen Polygonzug, wie es weiter vorne gezeigt wurde.

Da aber fast jede Messung statistisch gesehen eine Stichprobe ist, kann man durch diese Beobachtungspunkte eine Regressionsgerade legen. Diese Gerade wird so gelegt, daß die Summe der Quadrate aller Abstände der Punkte von der Geraden minimal wird.

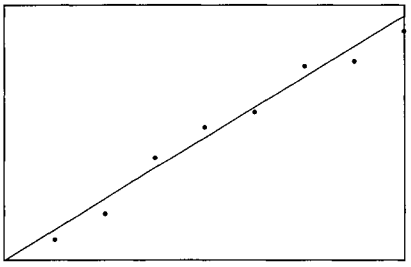


Abb. 4.1 Meßpunkte und Regressionsgerade

Dieses Verfahren nennt sich daher „Prinzip der kleinsten Quadrate". Zu jedem Punkt $\begin{pmatrix} x_i & y_i \end{pmatrix}, i = 1, \ldots, n$ gibt es den vertikalen Abstand vom Punkt der Geraden mit gleicher x-Koordinate:

$$r_i = y_i' - y_i \qquad i = 1, \ldots, n.$$

Die Summe der r_i^2 soll minimiert werden. Die Gleichung der Regressionsgeraden ist:

$$y' = A + B \cdot x.$$

Zu bestimmen sind also A und B. Setzt man die Geradengleichung in die Formel oben ein, erhält man:

$$\begin{aligned} r_i &= A + B \cdot x_i - y_i \qquad i = 1, \ldots, n \\ \Longrightarrow S := \sum_{i=1}^{n} r_i^2 &= \sum_{i=1}^{n} (A + B \cdot x_i - y_i)^2. \end{aligned}$$

Das Minimum der Summe erreicht man, indem die partielle Ableitung der Gleichung nach A und B gleich Null gesetzt werden. Die Differentiation ergibt:

$$\sum_{i=1}^{n} A + B \sum_{i=1}^{n} x_i = \sum_{i=1}^{n} y_i$$

und

$$A \cdot \sum_{i=1}^{n} x_i + B \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n} (x_i \cdot y_i).$$

Dieses lineare Gleichungssystem läßt sich leicht nach A und B auflösen. So lassen sich die Koeffizienten der Regressionsgeraden ausrechnen.

Bei allen Approximationsverfahren stellt sich die Frage, wie genau die Annäherung an die Meßwerte ist. Daher wird zusätzlich ein Korrelationskoeffizient berechnet, der angibt, wie die Standardabweichung der gegebenen Werte mit der Standardabweichung der approximierten Geraden übereinstimmt. Je näher der Betrag des Koeffizienten bei 1 liegt, desto besser liegt unsere Regressionsgerade. Da die meisten Werte für die Berechnung des Korrelationskoeffizienten sowieso vorliegen, ist der Rechenaufwand dafür recht gering:

$$K = \frac{\sum_{i=1}^{n}(x_i \cdot y_i) - \frac{1}{n}\sum_{i=1}^{n} x_i \cdot \sum_{i=1}^{n} y_i}{\left(\sum_{i=1}^{n}(x_i \cdot x_i) - \frac{1}{n}(\sum_{i=1}^{n} y_i)^2\right) \cdot \left(\sum_{i=1}^{n}(y_i \cdot y_i) - \frac{1}{n}(\sum_{i=1}^{n} y_i)^2\right)}.$$

4.3 Logarithmische Approximation

Nicht jede Meßpunktreihe hat einen linearen Verlauf. Ein weiterer Ansatz wäre die logarithmische Funktion

$$y = A + B \cdot \ln(x).$$

Die Ermittlung der Parameter A und B erfolgt nach dem gleichen Strickmuster wie bei der linearen Approximation:

$$\begin{aligned} B &= \frac{\sum_{i=1}^{n}(\ln(x_i)\cdot y_i) - \frac{1}{n}(\sum_{i=1}^{n}\ln(x_i)\cdot\sum_{i=1}^{n} y_i)}{(\sum_{i=1}^{n}\ln(x_i))^2 - \frac{1}{n}\sum_{i=1}^{n}\ln(x_i)^2} \\ A &= \frac{1}{n}\sum_{i=1}^{n} y_i - \frac{B}{n}\cdot\sum_{i=1}^{n}\ln(x_i). \end{aligned}$$

Auch der Korrelationskoeffizient wird auf die gleiche Weise wie oben berechnet.

4.4 Exponentielle Approximation

Während bei der linearen Approximation die Steigung der Kurve gleich bleibt und beim logarithmischen Typ abnimmt, hat sie bei der exponentiellen Approximation eine ständige Zunahme. Der Lösungsansatz ist hier:

$$y = A + e^{Bx}.$$

Das Berechnungsschema ist nach dem bewährten Muster geschneidert. Die Bestimmungsgleichung für A und B entsprechen wieder jenen der linearen Approximation:

$$\begin{aligned} B &= \frac{\sum_{i=1}^{n}\ln(y_i) - \frac{1}{n}\sum_{i=1}^{n} x_i\cdot\sum_{i=1}^{n}\ln(y_i)}{\sum_{i=1}^{n} x_i^2 - \frac{1}{n}(\sum_{i=1}^{n} x_i)^2} \\ A &= e^{(\sum_{i=1}^{n}(\ln(y_i)) - B\cdot\sum_{i=1}^{n} x_i)}. \end{aligned}$$

4.5 Gleitender Durchschnitt

Vielfach decken sich die Meßpunkte weder mit der Regressionsgeraden noch mit einem logarithmischen oder exponentiellen Verlauf. Besonders bei saisonal schwankenden Wirtschaftsdaten ergibt sich oft der in Abb. 4.2 gezeigte Verlauf von Meßpunkten. Derart oszillierende Daten lassen oft erst einen Trend erkennen, wenn man jeweils den Mittelwert aus 3 bis 9 Werten bildet und diesen dann aufträgt.

Für den Punkt P_i nimmt man die Werte, deren Index im Bereich

$$-K \text{ DIV } 2, \ldots, +K \text{ DIV } 2$$

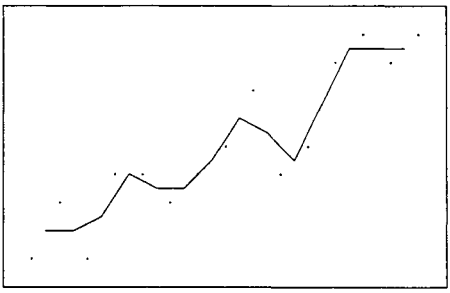


Abb. 4.2 Meßpunkte durch gleitenden Durchschnitt geglättet

liegt. K ist dabei die Zahl der zu mittelnden Werte. K muß in jedem Fall ungerade sein. Es fallen natürlich die ersten und letzten $K/2$ Werte weg. Da für die Berechnung der Mittelwerte immer die Nachbarwerte eines y_i genommen werden, „gleitet" der Mittelwert durch alle Punkte; Trends werden offenbar. Die einzelnen Punkte werden z.B. für $K = 3$ nach folgendem Schema berechnet:

$$\begin{aligned} y_2 &= \frac{(y_1 + y_2 + y_3)}{3} \\ y_3 &= \frac{(y_2 + y_3 + y_4)}{3} \\ &\vdots \\ y_{n-1} &= \frac{(y_{n-2} + y_{n-1} + y_n)}{3}. \end{aligned}$$

4.6 Polynom-Interpolation

Bei den meisten Aufgaben der grafischen Datenverarbeitung besteht das Problem darin, eine relativ geringe Menge von Stützpunkten durch eine möglichst glatte Kurve zu verbinden.

Man braucht also ein Verfahren, das zu den gegebenen Stützpunkten beliebig viele Zwischenwerte berechnet. Die bisherigen Approximationsverfahren scheiden hier aus, da bei diesen Methoden die Stützpunkte fast nie in der Kurve enthalten sind. Mit der Newton-Interpolation kann man zu einer Menge von Stützpunkten $\begin{pmatrix} x_i & y_i \end{pmatrix}$ ein Polynom angeben, dessen Grad gleich der Anzahl der Stützpunkte minus 1 $(n-1)$ ist:

$$\begin{aligned} Q_n(x) &= a_o + a_1(x - x_1) + a_2(x - x_1)(x - x_2) + \ldots \\ &\quad + a_n(x - x_1)(x - x_2)\ldots(x - x_n). \end{aligned}$$

Die Koeffizienten $a_o, \ldots, a_n$ lassen sich mit dem Newton-Verfahren explizit berechnen. Sie werden mit Hilfe der dividierten Differenzen

$$f(x_i, x_{i+1}, \ldots, x_{i+n})$$

ermittelt. Diese sind rekursiv definiert durch:

$$\begin{array}{rcll} f(x_i) & = & y_i & i = 1, \ldots, n \\ f(x_i, x_{i+1}) & = & \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} & i = 1, \ldots, n \end{array}$$

und so weiter. Die Koeffizienten a_i werden dann durch

$$a_i = f(x_0, x_1, \ldots, x_i)$$

bestimmt. Sie werden also rekursiv berechnet; das Interpolationspolynom kann dann mit dem Hornerschema berechnet werden.

Einen Nachteil hat die Polynom-Interpolation. Wenn die Stützwerte stark oszillieren, dann zeigt die interpolierende Kurve starke „Überschwinger", besonders am Anfang und am Ende und bei vielen Stützpunkten. Wie sich dieses Problem lösen läßt, zeigt der folgende Abschnitt.

4.7 Spline-Interpolation

Einen glatten Verlauf der interpolierenden Kurve erhält man, wenn man zwischen den Stützstellen mit kubischen Polynomen interpoliert. Das Polynom hat also den Grad 3, es muß zweimal stetig differenzierbar sein und als Randbedingung gilt, daß die zweiten Ableitungen der Randpunkte Null sein müssen:

$$S''(x_0) = 0 \qquad \text{und} \qquad S''(x_n) = 0.$$

Die Stützpunkte $\begin{pmatrix} x_i & y_i \end{pmatrix}$ müssen geordnet sein, also

$$x_0 < x_1 < \ldots < x_{n-1} < x_n.$$

Man kann sich das Verfahren bildlich so vorstellen, daß ein elastisches Stahlband an den Stützpunkten befestigt wird. Auch im Straßenbau verwendet man für Einmündungen und Abzweigungen kubische Splines, damit die Straßen „glatt" ineinanderlaufen. Betrachten wir das Intervall von x_i bis x_{i+1}, dann gilt der Ansatz:

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3.$$

Es ergeben sich so die folgenden Bedingungen.

$$\begin{array}{rcll} S_i(x_i) & = & y_i & i = 1, \ldots, n \\ S_i(x_i) & = & S_{i-1}(x_i) & i = 1, \ldots, n-1 \\ S_i'(x_i) & = & S_{i-1}'(x_i) & i = 1, \ldots, n-1 \\ S_i''(x_i) & = & S_{i-1}''(x_i) & i = 1, \ldots, n-1 \end{array}$$

Unter Berücksichtigung dieser Bedingungen kann man für die Koeffizienten der kubischen Splines folgendes Gleichungssystem aufstellen:

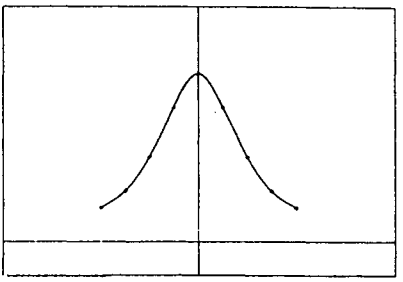


Abb. 4.3 Spline-Interpolation

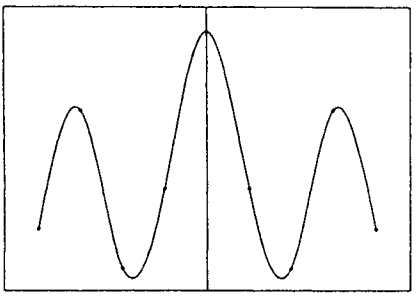


Abb. 4.4 Spline-Interpolation bei stark schwankenden Werten

$$
\begin{aligned}
a_i &= y_i \qquad i = 0, \ldots, n \\
c_0 &= 0 \qquad \text{(Randbedingung)} \\
c_n &= 0 \qquad \text{(Randbedingung)} \\
b_i &= \frac{(a_{i+1} - a_i)}{(x_{i+1} - x_i)} - 3(c_{i+1} + 2c_i)(x_{i+1} - x_i) \\
d_i &= \frac{(c_{i+1} - c_i)}{3(x_{i+1} - x_i)}.
\end{aligned}
$$

Die Koeffizienten c_i lassen sich aus einem tridiagonalen, linearen Gleichungssystem berechnen.

Mit $h_i = x_{i+1} - x_i$ gilt:

$$
c_{i-1} \cdot h_i + 2 \cdot c_i(h_{i-1} + h_i) + c_{i+1}h_i = \frac{3(a_{i+1} - a_i)}{h_i} - \frac{3(a_i - a_{i-1})}{h_{i-1}}.
$$

Die Matrix heißt Tridiagonalmatrix, weil nur die Hauptdiagonale und die beiden Nebendiagonalen besetzt sind. Das so entstandene Gleichungssystem läßt sich mit relativ wenig Aufwand lösen.

Abb. 4.3 zeigt, wie sich die Splinefunktion an die Stützpunkte anschmiegt. Abb. 4.4 beweist, daß sich die Spline-Interpolation auch bei einem üblen Stützpunktwirrwarr ausgesprochen „gutmütig“ verhält.

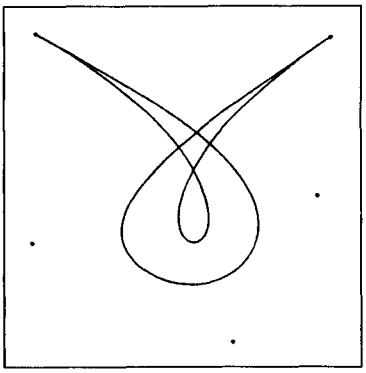


Abb. 4.5 Bézierkurve mit und ohne Mehrfachpunkte

4.8 Bézier-Kurven

Sieht man sich den Befehlsvorrat von grafikfähigen Computern genauer an, so wird man feststellen, daß es meist nur Grafikbefehle gibt, die entweder einfache Punkte setzen oder aber gerade Linien als Verbindung zwischen Koordinatenpaaren ziehen können. Durch die Methode von Bézier können beliebig gekrümmte Kurven entwickelt werden, ohne eine mathematische Funktion bestimmen zu müssen.

Das Problem ist nur, wie sind beliebige Kurven zu definieren? Für jede gewünschte Kurve eine mathematische Funktion zu bestimmen, ist so gut wie unmöglich. Man braucht also eine möglichst einfache Methode, gekrümmte Linien interaktiv zu formen. P. Bézier entwickelte für die französische Autofirma Renault ein System zur Gestaltung gekrümmter Automobilteile mittels Computer-Grafik.

Die Bézier-Kurve verhält sich ähnlich wie ein Stahlband, das zwischen zwei Punkten eingespannt ist (wobei man einmal annehmen möge, daß es auch noch dehnbar wie ein Gummiband sei). Wenn man nun in der Mitte das Stahlband nach außen zieht, wird es sich verformen, aber wegen seiner Elastizität wird es keinen Knick machen oder brechen, sondern eine Ausbeulung zeigen. Nun könnte man noch weitere Zugpunkte an dem Band anbringen, die an unterschiedlichen Stellen in verschiedenen Richtungen und mit anderer Kraft ziehen.

Auf diese Art könnte man beliebige Kurven gestalten, die keine Ecken aufweisen würden. Um bei diesem Beispiel zu bleiben: Bei der Bézier-Kurve bilden die Kontrollpunkte die Stellen, mit denen die Kurven geformt werden, und die „Zugkraft" wird durch die Position der Kontrollpunkte und ihrer Entfernung zur Kurve sowie durch Zusammenlegen mehrerer Punkte zu einem sogenannten Mehrfachpunkt oder Multi-Punkt bestimmt.

Bis auf die Endpunkte liegen alle Kontrollpunkte außerhalb der Kurve (es sei denn, die Kurve geht genau durch einen hindurch). Eine Eigenschaft der

Bézier-Kurve ist, daß sich bei Änderung eines Kontrollpunktes dieses auf die gesamte Kurve überträgt, man spricht hier von globalem Einfluß.

Mathematisch gesehen wird die Kurve berechnet, indem der Einfluß der Kontrollpunkte auf eine beliebig große Zahl von Kurvenpunkten nach einer bestimmten Funktion gewichtet wird. Diese Gewichtung erfolgt nach einer statistischen Verteilung, die für jeden Kontrollpunkt den entsprechenden Einfluß auf die Kurvenform festlegt. So ist der Einfluß in der Nähe am größten. Und auf die Endpunkte hat ein Kontrollpunkt in der Mitte überhaupt keinen Einfluß, denn diese liegen für ihn fest. In der unteren angegebenen Parameter-Darstellung der Gleichung zur Bestimmung der Bézier-Kurve ist folgendes zu erkennen:

Für jeden Kurvenpunkt c, von denen es beliebig viele gibt, je nach Schrittweite, wird der Einfluß aller Kontrollpunkte k von $i = 0$ bis n (aber: Anzahl aller Kontrollpunkte $= n+1$) nach einer Funktion bestimmt, welche in Abhängigkeit vom jeweils nächstliegenden Kontrollpunkt nach einer Binominal-Verteilung den Kurvenwert aufsummiert. Die Inkrementierung des Parameters u läuft von 0 bis 1 und bestimmt die Anzahl der Kurvenpunkte.

Dazu wird der Binominal-Koeffizient

$$B_i = \frac{n!}{i! \cdot (n-i)!} \qquad i = 0, \ldots, n$$

berechnet und anschließend der Ausdruck in

$$C(u) = \sum_{i=1}^{n} k_i \cdot B_i \cdot u^i \cdot (1-u)^{n-i} \qquad 0 \leq u \leq 1$$

eingesetzt.

4.9 Stereobilder

Wir Menschen können dreidimensional sehen und so zum Beispiel Entfernungen schätzen, weil wir zwei Augen besitzen. Man muß für stereoskopische Bilder also zwei perspektivische Bilder erzeugen, deren Betrachterabstände sich geringfügig unterscheiden.

Bei der Bildschirmausgabe stellt sich das Problem, daß jedes Auge nur ein Bild wahrnehmen darf. Eine Lösung wäre die Filtermethode, die auch bei Plotterzeichnungen angewendet werden kann. Der Betrachter trägt eine Brille mit farbigen Filtergläsern, Rot für das linke Auge, Grün für das rechte Auge. Der Computer erzeugt zwei Bilder: ein grünes für das linke Auge und ein rotes für das rechte Auge. Da die Filtergläser genau die umgekehrte Farbe haben, sieht jedes Auge nur das für dieses Auge bestimmte Bild. Eine weitere Möglichkeit wäre, die Bilder für das rechte und das linke Auge abwechselnd mit einer Bildwechselfrequenz von mehr als 25 Hertz darzustellen und durch ein umschaltbares Filter vor dem Bildschirm unterschiedlich zu polarisieren.

Mit einer geeigneten Polarisationsbrille kann dann auch dreidimensional gesehen werden. Eine weitere Möglichkeit wird durch eine Brille mit LCD-Shutter

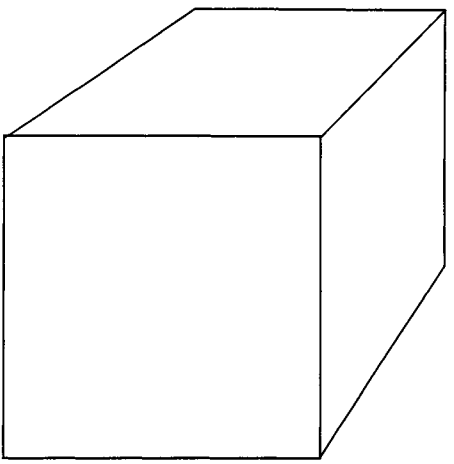
geboten, die wechselweise ein Auge abdeckt (das Glas ist entweder durchsichtig oder schwarz) und mit dem Bild synchronisiert wird. Solche Brillen werden für Computerspiele bereits angeboten. Schließlich bleibt als aufwendigste Lösung die Positionierung zweier getrennter Miniaturmonitore direkt vor den Augen, eine Lösung, die auch bereits realisiert wurde. Bei Zeichnungen kann man zum Betrachten der Stereobilder auch ein Stereoskop verwenden. Dazu werden zwei Bilder (in schwarzweiß oder Farbe) erzeugt, die dann mit dem Stereoskop betrachtet werden. Das Stereoskop besitzt für jedes Auge ein eigenes optisches System (wie es auch beim Feldstecher der Fall ist), so daß auch hier jedes Auge nur eines der Bilder sehen kann.

Der Betrachtungsstandpunkt wird für die beiden Bilder um den gleichen Betrag in positiver und negativer x-Richtung verschoben. Der Punkt $\begin{pmatrix} x & y & z \end{pmatrix}$ geht dann einmal in $\begin{pmatrix} x+e & y & z \end{pmatrix}$ und zum anderen in $\begin{pmatrix} x-e & y & z \end{pmatrix}$ über. Weitere Einzelheiten zum Berechnen und Betrachten von Stereobildern finden Sie in Abschnitt 20.3.

II

Grundlagen 3D-Algorithmen und Rendering

Ute Claussen
Josef Pöpsel



5. Mathematische Grundlagen

Für die Darstellung dreidimensionaler Körper ist eine Erweiterung der mathematischen Grundlagen notwendig. Punkte und Vektoren werden in einer einheitlichen Notation, den homogenen Koordinaten, dargestellt. Lineare Abbildungen und ihre Repräsentation als Matrizen ermöglichen Transformationen verschiedenster Art, unter anderem perspektivische Darstellungen.

5.1 Punkte und Vektoren

Punkte in einer dreidimensionalen „Welt" werden durch drei reelle Komponenten beschrieben. Sie werden im allgemeinen als x, y und z bezeichnet und als Spaltenvektor

$$P = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

oder als Zeilenvektor

$$P = \begin{pmatrix} x & y & z \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}^T$$

dargestellt. Das Zeichen „T" hat den Namen „transponiert" und bedeutet ein Vertauschen von Zeilen und Spalten. Es kann auch auf Matrizen angewendet werden (siehe auch im nächsten Abschnitt).

Im Gegensatz zu einer Position, die durch einen Punkt beschrieben wird, hat ein *Vektor* eine Richtung und eine Länge. Verwirrenderweise werden für Vektoren und Punkte gleiche Notationen verwendet:

$$V = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Wir wollen im folgenden Vektoren durch Fettdruck von den Punkten unterscheiden, also:

$$\mathbf{V} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}.$$

Die Richtung des Vektors ist durch x, y und z gegeben, seine Länge durch

$$|\mathbf{V}| = \sqrt{x^2 + y^2 + z^2}.$$

Vektoren der Länge 1 werden als normierte Vektoren bezeichnet. Sie spielen eine besondere Rolle, z.B. bei Beleuchtungsberechnungen.

Ein Vektor kann auch als die Differenz zweier Punkte $P_1 = \begin{pmatrix} x_1 & y_1 & z_1 \end{pmatrix}$ und $P_2 = \begin{pmatrix} x_2 & y_2 & z_2 \end{pmatrix}$ interpretiert werden. Ist er als

$$\mathbf{V} = P_2 - P_1 = \begin{pmatrix} x_2 - x_1 \\ y_2 - y_1 \\ z_2 - z_1 \end{pmatrix}$$

definiert, so zeigt er von P_1 in Richtung P_2 und hat die Länge des Abstandes zwischen P_1 und P_2.

Ebenso wie im zweidimensionalen können dreidimensionale Punkte und Vektoren homogen erweitert werden. Diese Erweiterung hat mehrere Vorteile:

1. Punkte und Vektoren können in einer einheitlichen Notation zusammengefaßt werden.
2. Alle Transformationen, die für die Darstellung von Szenen benötigt werden, insbesondere auch perspektivische Transformationen, können einheitlich notiert werden.
3. Die Notation schließt auch die Beschreibung von Kurven und Flächen dritter Ordnung mit ein und ist damit ein allgemein einsetzbares Hilfsmittel.

Die homogene Erweiterung eines dreidimensionalen Punktes in Spaltendarstellung ist

$$P_h = \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix}$$

und in Zeilendarstellung

$$P_h = \begin{pmatrix} 1 & x & y & z \end{pmatrix}.$$

Allgemein also:

$$P_h = \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} \quad \text{bzw.} \quad P_h = \begin{pmatrix} w & x & y & z \end{pmatrix}.$$

Die Rückführung der homogenen in die kartesischen Koordinaten besteht in der Division durch die w-Komponente, also

$$P = \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \quad \text{bzw.} \quad P = \begin{pmatrix} x/w & y/w & z/w \end{pmatrix}.$$

Der Fall $w = 0$ hat eine besondere Bedeutung, wie wir gleich sehen werden, und muß daher in einem Programm auf alle Fälle abgefangen werden.

Typischerweise wird ein Punkt als Position mit den Koordinaten x, y und z definiert, anschließend um die homogene Koordinate erweitert, Transformationen unterzogen und dann in kartesische Koordinaten rücktransformiert.

$$\begin{array}{ccc} P & \xrightarrow{\text{homogene Erweiterung}} & P_h \\ & & \downarrow \text{homogene Transformation(en)} \\ P' & \xleftarrow{\text{Division durch } w} & P'_h \end{array}$$

Die homogene Erweiterung eines Vektors ergibt sich aus der Interpretation als Differenz zweier Punkte.

$$\mathbf{V}_h = P_{2h} - P_{1h} = \begin{pmatrix} 1 \\ x_2 \\ y_2 \\ z_2 \end{pmatrix} - \begin{pmatrix} 1 \\ x_1 \\ y_1 \\ z_1 \end{pmatrix} = \begin{pmatrix} 0 \\ \mathbf{V} \end{pmatrix}$$

Vektoren werden also mit $w = 0$ erweitert.

5.2 Matrizen

Matrizen sind eine Form der Repräsentation linearer Abbildungen. Sie werden im dreidimensionalen Fall als

$$M = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}$$

notiert und im vierdimensionalen, homogenen Fall als

$$M = \begin{pmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{pmatrix}.$$

Matrizen haben, je nach der Abbildung, die sie ausdrücken, bestimmte Eigenschaften, die später beschrieben werden.

5.3 Vektor- und Matrizenalgebra

Für die Rechnung mit Vektoren und Matrizen gelten bestimmte Regeln, die hier noch einmal zusammengefaßt werden sollen.

Die *Transponierte* einer Matrix ist die Matrix, in der Zeilen und Spalten vertauscht sind.

$$M^T = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}^T = \begin{pmatrix} a_{11} & \cdots & a_{n1} \\ \vdots & & \vdots \\ a_{1m} & \cdots & a_{nm} \end{pmatrix}$$

Gleiches gilt für Vektoren.

$$\mathbf{V}^T = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}^T = \begin{pmatrix} x_1 & \cdots & x_n \end{pmatrix}$$

Die *Inverse* einer Matrix, die auch gleichzeitig die inverse Abbildung beschreibt, ist über

$$M^{-1} \cdot M = M \cdot M^{-1} = E = \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix}$$

definiert. Sie ist nur für $n = m$ definiert und kann z.B. durch die folgenden Beziehungen berechnet werden:

$$M^{-1} = \frac{1}{\det M} \begin{pmatrix} M_{11} & M_{12} & \cdots & M_{1n} \\ \vdots & & & \vdots \\ M_{n1} & M_{n2} & \cdots & M_{nn} \end{pmatrix}^T$$

mit

$$M_{ij} = (-1)^{i+j} \det \begin{pmatrix} a_{11} & \cdots & a_{1,j-1} & a_{1,j+1} & \cdots & a_{1n} \\ \vdots & & & & & \vdots \\ a_{i-1,1} & & & & & a_{i-1,n} \\ a_{i+1,1} & & & & & a_{i+1,n} \\ \vdots & & & & & \vdots \\ a_{n1} & \cdots & a_{n,j-1} & a_{n,j+1} & \cdots & a_{nn} \end{pmatrix}.$$

M_{ij} ist die $(n-1 \times n-1)$-Matrix, die aus der Matrix M durch Streichen der i-ten Zeile und der j-ten Spalte entsteht. $\det M$ bezeichnet die Determinante einer Matrix. Für $n = 2$ und $n = 3$ ist

$$\det M = \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} = a_{11}a_{22} - a_{21}a_{12}$$

und

$$\begin{aligned} \det M &= \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} \\ &= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} \\ &\quad -a_{13}a_{22}a_{31} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33}. \end{aligned}$$

Für größere Matrizen, z.B. die homogene 4×4-Matrix, kann die Determinante über den Entwicklungssatz berechnet werden (siehe z.B. Bronstein [BS81]).

Die Hintereinanderausführung von Abbildungen entspricht einer Multiplikation der zugehörigen Matrizen. Das Produkt $A \cdot B$ zweier Matrizen A und B ist

$$C = A \cdot B \qquad \text{mit} \qquad c_{ik} = \sum_{j=1}^{n} a_{ij} b_{jk}, \qquad 1 \leq i, k \leq n,$$

d.h. das Ergebnis c_{ik} ist die Summe der Produkte der i-ten Zeile von A und der j-ten Spalte von B. Voraussetzung für die Anwendbarkeit dieser Operation ist die Übereinstimmung der Spaltenzahl von A mit der Zeilenzahl von B.

Die Multiplikation kann auch auf das Produkt von Matrix und Spaltenvektor angewendet werden. Das Resultat ist dann der durch die Abbildung transformierte Vektor oder Punkt:

$$P' = M \cdot P.$$

Das transponierte Produkt zweier Matrizen ist:

$$C^T = (A \cdot B)^T = B^T \cdot A^T.$$

Das bedeutet für die Notation der Punkte und Vektoren in Zeilenform, daß sie „von vorne" an die Matrix multipliziert werden

$$P'^T = P^T \cdot M^T.$$

Einen weiteren Vorteil hat die Notation mit Zeilenvektoren bei mehrfachen Abbildungen. Wird auf einen Spaltenvektor zuerst M_1 und dann M_2 angewendet, so gilt:

$$P' = M_2 \cdot M_1 \cdot P,$$

d.h. die Matrizen müssen in umgekehrter Reihenfolge miteinander verknüpft werden. Im anderen Fall gilt:

$$P'^T = (M_2 \cdot M_1 \cdot P) = P^T \cdot M_1^T \cdot M_2^T.$$

Die Matrizen werden also in der Reihenfolge, in der sie auch auftreten, „an den Punkt" multipliziert.

5.4 Transformationen

Transformationen sind der „Anwendungsfall" für Matrizen in der Computergrafik. Sie dienen z.B. dazu, Punkte und damit ganze Objekte von einer lokalen in eine globale „Welt" zu transformieren, also das Koordinatensystem zu wechseln. Im folgenden werden die gebräuchlichsten Formen dargestellt und die Spezialfälle der Transformation von Normalenvektoren und Koordinatensystemen besprochen.

Für die folgenden Darstellungen soll der Einfachheit halber mit Zeilenvektoren gearbeitet werden. Sie sind teilweise ungewohnt, führen aber zur verwendeten Notation der Programme.

5.4.1 Transformation von Punkten und Vektoren

Die zweidimensionalen Transformationen werden im folgenden auf homogene dreidimensionale Koordinaten erweitert.

Eine *Skalierung* wird durch

$$\begin{aligned} x' &= a \cdot x \\ y' &= b \cdot y \\ z' &= c \cdot z \end{aligned}$$

und damit durch die Operation

$$P' = \begin{pmatrix} w' & x' & y' & z' \end{pmatrix} = \begin{pmatrix} w & x & y & z \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & b & 0 \\ 0 & 0 & 0 & c \end{pmatrix}$$

beschrieben. Im Fall, daß $a = b = c$ gilt, kann die Skalierung auch durch eine sogenannte Gesamtskalierung

$$w' = \frac{1}{a} \cdot w$$

erreicht werden. Beim Übergang von homogene in kartesische Koordinaten wird damit z.B. die x-Komponente

$$x_k = \frac{x_h}{w'} = x_h \cdot a.$$

Die *Translation*, also die Verschiebung um einen Vektor $\begin{pmatrix} a & b & c \end{pmatrix}$, wird durch

$$\begin{aligned} P' &= \begin{pmatrix} w' & x' & y' & z' \end{pmatrix} \\ &= \begin{pmatrix} 1 & x & y & z \end{pmatrix} \cdot \begin{pmatrix} 1 & a & b & c \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & x+a & y+b & z+c \end{pmatrix} \end{aligned}$$

dargestellt.

Eine *Rotation* kann im dreidimensionalen Raum um drei Achsen vorgenommen werden: um die x-, die y- und um die z-Achse. Die Rotationsmatrizen entstehen durch entsprechende Umstellung. Eine Rotation um die x-Achse um den Winkel α beschreibt

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \cos\alpha & -\sin\alpha \\ 0 & 0 & \sin\alpha & \cos\alpha \end{pmatrix}.$$

Eine Rotation um die y-Achse um den Winkel β zeigt

$$R_y = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\beta & 0 & -\sin\beta \\ 0 & 0 & 1 & 0 \\ 0 & \sin\beta & 0 & \cos\beta \end{pmatrix}.$$

Schließlich beschreibt eine Rotation um die z-Achse um den Winkel γ:

$$R_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma & 0 \\ 0 & \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

Weitere, allgemeine Transformationen entstehen durch Hintereinanderausführung, d.h. Multiplikation, bzw. Besetzen der anderen Positionen der Matrix.

5.4.2 Transformation von Normalenvektoren

Normalenvektoren sind Vektoren (meist mit Einheitslänge), die auf einer Fläche senkrecht stehen. Für die Beleuchtungsberechnung und das Entfernen der Rückflächen werden transformierte Normalenvektoren benötigt. Damit sie ihre Eigenschaften behalten und nach der Transformation wieder senkrecht auf der transformierten Fläche stehen, dürfen sie allerdings nicht mit der gleichen Matrix wie die Fläche, sondern müssen mit der Transponierten der inversen Matrix abgebildet werden. Das wird im folgenden gezeigt.

Für jeden Vektor $\mathbf{V}$ auf einer gegebenen Ebene und der Normale $\mathbf{N}$ der Ebene verschwindet das Skalarprodukt zwischen den beiden Vektoren.

$$(\mathbf{V} \cdot \mathbf{N}) = \mathbf{V} \cdot \mathbf{N}^T = 0$$

Der Vektor wird durch eine Transformation M auf einen zweiten abgebildet.

$$\mathbf{V}' = \mathbf{V} \cdot M$$

Es ist nun die Matrix M' gesucht, so daß $\mathbf{N}$ dadurch zu $\mathbf{N}'$ transformiert wird:

$$\mathbf{N}' = \mathbf{M}' \cdot N$$

und das Skalarprodukt zwischen $\mathbf{N}'$ und $\mathbf{V}'$ wieder identisch Null ist.

$$(\mathbf{V}' \cdot \mathbf{N}') = \mathbf{V}' \cdot \mathbf{N}'^T = 0$$

Ausgehend von dem ersten Skalarprodukt gilt:

$$\begin{aligned} & & \mathbf{V} \cdot \mathbf{N}^T &= 0 \\ \Leftrightarrow & & \mathbf{V} \cdot M \cdot M^{-1} \cdot \mathbf{N}^T &= 0 \\ \Leftrightarrow & & \mathbf{V}' \cdot ((M^{-1} \cdot \mathbf{N}^T)^T)^T &= 0 \\ \Leftrightarrow & & \mathbf{V}' \cdot (\mathbf{N} \cdot (M^{-1})^T)^T &= 0 \\ & \Leftrightarrow & \mathbf{V}' \cdot \mathbf{N}'^T &= 0 \\ & \Rightarrow & \mathbf{N}' &= \mathbf{N} \cdot (M^{-1})^T \\ & \Rightarrow & M' &= (M^{-1})^T, \end{aligned}$$

also wird ein Normalenvektor mit der Transponierten der Inversen transformiert.

5.4.3 Transformation von Koordinatensystemen

Ohne Beweis oder nähere Begründung soll die Transformation von Koordinatensystemen angegeben werden. Sie läßt sich in Grundlagenwerken zur linearen Algebra finden. Für homogene Koordinaten $\begin{pmatrix} w & x & y & z \end{pmatrix}$, die von einem Koordinatensystem A in ein Koordinatensystem B transformiert werden sollen, setzt sich die Transformationsmatrix M_{A_B} zeilenweise aus den Basisvektoren zusammen.

$$M_{A_B} = \begin{pmatrix} A_w \\ A_x \\ A_y \\ A_z \end{pmatrix}$$

A_w ist die Koordinate des Ursprungs von A, ausgedrückt in den Koordinaten des Systems B. A_x, A_y und A_z sind die Koordinaten der Basisvektoren von A, ebenfalls ausgedrückt in B.

Eventuell kann es einfacher sein, M_{B_A} zuerst zu bestimmen und dann zu invertieren. Beispiele der Bestimmung von Transformationen zwischen den Koordinatensystemen finden sich im Abschnitt zu den Koordinatensystemen bei der Darstellung dreidimensionaler Körper und Flächen (Kap. 7.1).

5.4.4 Projektionen

Eine Projektion dient der Abbildung auf einen niedriger dimensionierten Raum. Eine solche Abbildung benötigen wir, wenn dreidimensionale Objekte auf dem zweidimensionalen Bildschirm dargestellt werden.

Im künstlerischen Bereich wie in der Computergrafik werden verschiedene Projektionen unterschieden. Die beiden Hauptklassen sind parallele und perspektivische Projektion. Sie unterteilen sich jeweils in weitere Unterklassen, z.B. 1-, 2- und 3-Punkt-Perspektive oder rechtwinklige und schiefwinklige Parallelprojektionen [ES88].

Wir stellen hier die zwei in der Computergrafik am häufigsten verwendeten Projektionen vor: Parallel- und perspektivische Projektion.

Eine *Parallelprojektion* bildet alle Punkte auf eine konstante x-, y- oder z-Ebene ab. Beispielsweise wird die Parallelprojektion auf die Ebene $x = a$ durch die Beziehung

$$\begin{aligned} P' &= \begin{pmatrix} w' & x' & y' & z' \end{pmatrix} \\ &= \begin{pmatrix} w & a & y & z \end{pmatrix} \end{aligned}$$

$$= \begin{pmatrix} w & x & y & z \end{pmatrix} \cdot \begin{pmatrix} 1 & a & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

abgebildet. Analog ergeben sich die Projektionsmatrizen für die Abbildung auf die Ebene $y = b$ zu

$$M_{y=b} = \begin{pmatrix} 1 & 0 & b & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und für die Abbildung auf die Ebene $z = c$ zu

$$M_{z=c} = \begin{pmatrix} 1 & 0 & 0 & c \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Eine *perspektivische Projektion* entspricht eher der menschlichen Wahrnehmung. Sie bildet weit entfernte Objekte kleiner auf den Bildschirm ab als nah liegende Objekte. Wird ein Punkt $P = \begin{pmatrix} x & y & z \end{pmatrix}$ von einem Betrachtungspunkt im Ursprung auf die Ebene $z = z_0$ projiziert, gilt nach dem Strahlensatz

$$x' = x\frac{z_0}{z}, \qquad y' = y\frac{z_0}{z} \qquad \text{und} \qquad z' = z_0.$$

Diese Transformation kann wieder mit homogenen Koordinaten ausgedrückt werden.

$$\begin{aligned} P' &= \begin{pmatrix} w' & x' & y' & z' \end{pmatrix} \\ &= \begin{pmatrix} \frac{z}{z_0} & x & y & z \end{pmatrix} \\ &= \begin{pmatrix} 1 & x & y & z \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{1}{z_0} & 0 & 0 & 1 \end{pmatrix}. \end{aligned}$$

Nach der Division durch w beim Übergang auf kartesische Koordinaten gelten dann die oben angegebenen Beziehungen. Vor der Division ist allerdings die Tiefeninformation z noch vorhanden. Dies wird z.B. für die Verdeckungsrechnung mit dem z-Buffer genutzt.

6. Beschreibung der Programme

Bevor weitere Grundlagen der dreidimensionalen Computergrafik dargestellt werden, soll kurz auf die für diesen Teil dem Buch beigelegte Software eingegangen werden. Mit ihrer Hilfe werden im folgenden immer wieder Beispiele zu einzelnen Sachverhalten präsentiert.

6.1 Möglichkeiten der Grafikprogrammierung

Sollen mittels eines Programms automatisch Grafiken auf dem Rechner erzeugt werden, so bestehen dazu verschiedene Möglichkeiten:

- Die Beschreibung eines Bildes in einer Programmiersprache unter Verwendung einer Grafikbibliothek. In einer solchen Bibliothek sind beispielsweise die Modellierung, die Transformation und die Darstellung der in ihr beschriebenen Objekte mit ihren Eigenschaften gegeben. Bekannte funktionale Beschreibungen sind z.B. GKS (Grafisches Kernsystem, DIN 66252, [fN84]), PHIGS (Programmer's Hierarchical Interface to Graphics Systems, ISO-Norm 1989, [vD88]) oder die kommerziellen Systeme GL (Graphics Library) von Silicon Graphics oder RenderMan von Pixar [Ups89]. Der Nachteil dieser Vorgehensweise ist die Abhängigkeit der Implementierung von einer bestimmten Programmiersprache. Für die genormten Schnittstellen werden beispielsweise zusätzlich die verschiedenen Implementierungen in verschiedenen Sprachen genormt.
- Die Verwendung einer Eingabesprache, die von einem Programm dann zu einem Bild „interpretiert" wird. Solche Eingabesprachen enthalten typischerweise eine Beschreibung der Geometrie, der Betrachtung und der Beleuchtungsverhältnisse und beschreiben außerdem die Art der Darstellung des Bildes. Ein Beispiel einer solchen Eingabesprache ist das Eingabeformat von VERA (Very Efficient Raytracing Algorithm, [LMS91]).

Daneben besteht natürlich auch die Möglichkeit des interaktiven Arbeitens.

Wir haben für die Beschreibung des Rendering- und des Radiosity-Programms und dessen Eingabe die letzte Möglichkeit gewählt. Mit Hilfe der „objektorientierten" grafischen Eingabesprache `GED` wird eine Szene beschrieben, deren Form der Darstellung teilweise in der Eingabedatei und teilweise durch Menusteuerung im Programm beeinflußt wird.

6.2 Prinzipielles zur Implementierung

Das Hauptprogramm, mit dem die im Buch dargestellten Rendering- und Radiosity-Bilder erzeugt wurden, hat den Namen `PRIM`. Es befindet sich, ebenso wie alle benötigten Pascal-Module und einige Beispieldateien als Quelldatei auf der Diskette. Unter anderem sind dabei auch die Dateien, um die im Buch dargestellten Bilder zu erzeugen. Zum Übersetzen benötigen Sie Turbo-Pascal Version 6.0 oder 7.0 im „real mode". Mit Turbo-Pascal Version 5.0 sollte es ebenfalls möglich sein, die benötigten Routinen zu übersetzen, da kein Gebrauch von den objektorientierten Erweiterungen gemacht wurde.

Bei der Implementierung haben wir versucht, die Portierung auf andere Compiler und andere Rechner möglichst einfach zu gestalten. Aus diesem Grund fehlen Pascal-„with"-Anweisungen ganz. Es hat sich nämlich gezeigt, daß gerade diese Anweisungen bei der Portierung von Pascal-Programmen auf C-Programme die größten Schwierigkeiten machen. Weiterhin existieren bis auf wenige Ausnahmen keine lokalen Prozeduren, so daß ein anderes Hindernis bei der Portierung in die Sprache „C" entfällt.

Bis auf die beiden Module `PRIMMEM` und `PRIMOUT` sind alle verwendeten Module hardwareunabhängig. `PRIMMEM`, dem ein eigener Abschnitt im Anhang gewidmet ist, verwaltet die beim z-Buffer-Verfahren unweigerlich benötigten Mengen von Speicher. In `PRIMOUT` sind Routinen implementiert, die auf die verwendete Grafikkarte zugreifen. Für nicht unterstützte Grafikkarten müssen lediglich die Prozeduren `GrafikAn, GrafikAus, SetzePixel` und `LesePixel` neu implementiert werden. So wird es nicht sehr schwer fallen, auch für exotische Grafikkarten eine Anpassung vorzunehmen.

Außer dem erwähnten Pascal-Compiler ist für die Ausführung noch ein XMS-Treiber nötig, wie er z.B. von Microsoft mit jedem Windows ab Version 3.0 ausgeliefert wird. Dieser XMS-Treiber namens `HIMEM.SYS` wird mit Hilfe der `CONFIG.SYS`-Datei beim Starten des Rechners geladen. Mit ihm ist es möglich, auf den Speicher eines PCs oberhalb der „normalen" 640 kByte zuzugreifen. Das Ganze funktioniert natürlich nur bei PCs, die über mehr als 640 kByte Speicher verfügen. Wieviel dieses zusätzlichen Speichers `PRIM` benutzt, hängt von folgenden Faktoren ab:

1. der Komplexität der zu beschreibenden Szene,
2. der Größe des Fensters, in dem die Szene dargestellt werden soll,
3. der Auflösung der beim Radiosity verwendeten Singleplane und
4. der Anzahl und Größe der verwendeten Texturen.

Wenn während der Berechnung mehr Speicher angefordert wird, als Ihr Rechner zur Verfügung stellt, so bricht das Programm mit einer Fehlermeldung ab. In diesem Fall gibt es drei Möglichkeiten, wie Sie verfahren können:

1. Sie besorgen sich zusätzlichen Speicher.
2. Sie verringern den benötigten Speicher, indem Sie einen oder mehrere der gerade aufgeführten Punkte verändern.

3. Sie erweitern die Unit PRIMMEM so, daß Teile des benötigten Speichers auf Ihre Festplatte ausgelagert werden.

Auf die zweite und dritte Möglichkeit wird in Abschnitt C noch näher eingegangen.

6.3 Die Bedienung des Hauptprogramms PRIM

Das Hauptprogramm ist in der Lage, Grafische Eingabe Dateien (GED), also Szenenbeschreibungen, die in der Syntax der in Anhang B beschriebenen Eingabesprache verfaßt sind, durch die verschiedenen vorgestellten Algorithmen darzustellen. Nach welchem Verfahren die Berechnung erfolgt, muß nach dem Start des Programms eingegeben werden. Hierzu dient ein entsprechender Dialog.

Wenn PRIM mit einem Parameter aufgerufen wird, also z.B. „PRIM TEST", so gilt der Parameter als darzustellende Datei. Als Endung ist nur „.GED" erlaubt, bei fehlender Endung wird „.GED" selbständig angefügt. Wenn der Parameter fehlt, werden Sie aufgefordert, einen Dateinamen einzugeben.

Anschließend erfolgt das Einlesen der Datei und die Übersetzung. Wenn die GED-Datei syntaktisch nicht korrekt ist, erscheint eine Fehlermeldung und das Programm bricht ab.

Im anderen Fall folgen einige Eingabeaufforderungen, die das zu verwendende Darstellungsverfahren und diverse, an anderer Stelle dieses Buches angesprochene Punkte betreffen. Für jede Angabe existiert eine Standard-Antwort, so daß es im Zweifelsfall ausreicht, mit RETURN zu bestätigen. Am Ende der Eingaben erscheint schließlich (hoffentlich) das gewünschte Bild.

Wir haben absichtlich auf schöne Menus und eine ausgefeilte Benutzungsschnittstelle verzichtet, da diese i.a. sehr hardwareabhängig und außerdem nicht Thema diese Buches sind.

Nach erfolgter Darstellung kann der Betrachtungsstandpunkt geändert werden oder das Bild als Windows „.BMP"-Datei oder Targa-Datei („.TGA") gespeichert werden.

6.4 Beispiele zu den mathematischen Grundlagen

Im letzten Kapitel haben Sie die mathematischen Grundlagen der dreidimensionalen Computergrafik wiederholt. Insbesondere Transformationen spielen eine große Rolle bei der Modellierung und Darstellung. Daher wollen wir mit diesen Grundlagen gleichzeitig beginnen, das Programm PRIM und dessen Eingabesprache GED kennenzulernen.

Eine der einfachsten Szenen, die mit GED beschreibbar sind, ist ein schlichter Würfel.

```
SZENE Beispiel1;
  WUERFEL;
END.
```

Pascal-Programmierer werden die Syntax dieses Programms sicher auf Anhieb verstehen. Groß geschriebene Worte werden im folgenden Schlüsselworte der Eingabesprache anzeigen.

Erstellen Sie eine Datei `BSP1.GED` mit dieser Beschreibung und rufen Sie

```
PRIM BSP1
```

auf. Zuerst erfolgt eine Abfrage, welchen Typ von Grafikkarte Sie besitzen. Falls keiner der angegebenen Typen in Ihrem Rechner installiert ist, so müssen Sie zuerst das Programm auf ihren Rechner anpassen, wie es in Anhang C beschrieben ist.

Ist diese Hürde überwunden, so meldet sich das Programm mit einer Fehlermeldung:

```
Fehler beim Einlesen von bsp1.GED: GED_VERSION erwartet,
SZENE gefunden

Zeile 1
SZENE Beispiel1;
```

Die erste Zeile gibt die genaue Fehlermeldung an, die zweite die Zeile der Eingabedatei, in der der Fehler aufgetreten bzw. entdeckt worden ist. In der dritten Zeile, der fehlerhaften Zeile der Eingabedatei, ist das fehlerhafte Wort farbig markiert. Somit haben Sie auch schon eine Fehlermeldung von `PRIM` beim Einlesen einer `GED`-Datei kennengelernt.

Korrekt muß unsere einfache Beispieldatei so lauten:

```
GED_VERSION 1;
SZENE Beispiel1;
  WUERFEL;
END.
```

Aus einsichtigen Gründen haben die `GED`-Dateien Versionen, ebenso wie das Programm, das sie einliest.

Starten Sie nun das Programm wieder mit der Beispieleingabe, so erhalten Sie die Meldung:

```
Datei bsp1.GED erfolgreich gelesen
Dereferenzierte Anzahl Punkte: 8
Dereferenzierte Anzahl Flaechen: 6
```

Sie haben nun – nach dem Einlesen – die Möglichkeit, die Berechnungsparameter des Programms einzustellen. Da es uns im Moment nur darum geht, die Auswirkungen von Transformationen zu beobachten, sollten Sie die folgenden Fragen nach der gewünschten Unterteilung und der Berechnung mit Hilfe des Radiosity-Verfahrens verneinen.

Es erscheint ein schwarzer Würfel auf einem blau schattierten Hintergrund.

Mit einer beliebigen Taste kommen Sie in ein Menu, das Sie mit der Eingabe von `(E)nde` verlassen können. Einen Teil der Menueinträge werden wir im folgenden noch kennenlernen.

Damit Sie – obwohl Sie die entsprechenden Kapitel noch nicht gelesen haben – nicht nur einen schwarzen Würfel, sondern einen beleuchteten Würfel sehen, fügen Sie vor die Szene eine Lichtquelle ein.

```
GED_VERSION 1;
GERICHTETELQ 1,1,1 1,1,-1;
SZENE Beispiel1;
  WUERFEL;
END.
```

Die Zahlentripel hinter der Angabe der gerichteten Lichtquelle geben deren Farbe und Richtung an. Lassen Sie nun `PRIM` mit der Beispieleingabedatei laufen, so erscheint der Würfel beleuchtet und ebenfalls schattiert.

Die Transformationen, deren Wirkung wir mit diesem Beispiel beobachten wollten, werden direkt vor dem Würfel angegeben. Eine Gesamtskalierung des Würfels um den Faktor 2 geben Sie beispielsweise mit der folgenden Eingabe an.

```
GED_VERSION 1;
GERICHTETELQ 1,1,1 1,1,-1;
SZENE Beispiel1;
  SKALIERE 2;
  WUERFEL;
END.
```

Lassen Sie den Würfel nun wieder mit `PRIM` darstellen, so erscheint er in jeder seiner Ausdehnungen um den Faktor 2 vergrößert. Mit Hilfe der weiteren Angaben `VERSCHIEBE` und `DREHE` können Sie Translationen oder Rotationen angeben. Mit Hilfe von Kommentarzeilen wie in C++ (`//`) können Sie einzelne der folgenden Transformation ein- und ausschalten. Experimentieren Sie selbst – es kommt auf die Reihenfolge an!

```
GED_VERSION 1;
GERICHTETELQ 1,1,1 1,1,-1;
SZENE Beispiel1;
  SKALIERE 2;
  // VERSCHIEBE 3,0,0;
  // DREHE X,60;
  // DREHE Y,60;
  // DREHE Z,120;
  WUERFEL;
END.
```

Haben Sie die Transformationen und ihre Auswirkungen ausreichend beobachtet, so können Sie mit Hilfe des Menus nach der Darstellung des Bildes die Blickparameter verändern. Versuchen Sie beispielsweise

```
(W)ireframe darstellen
```

so werden anschließend nur die begrenzenden Kanten des Würfels dargestellt, ohne ihn zu schattieren.

Mit dem Eintrag

```
(Z)oom
```

vergrößern oder verkleinern Sie die Bildbreite. Kleinere Bildbreiten führen zu größeren Objekten, größere dagegen zu kleineren Objekten.

Ein weiterer Eintrag,

```
(P)erspektive aendern
```

bewirkt die Verschiebung des Bildabstands und somit der perpektivischen Verzerrung. Auch mit diesem Parameter können Sie experimentieren und so kennenlernen, was mit dem Programm darstellbar ist. Die hier nicht vorgestellten Parameter werden in den folgenden Kapiteln noch behandelt werden.

7. Grundlagen der dreidimensionalen Computergrafik

In den ersten Kapiteln dieses Buches wurden Grundlagen der zwei- und dreidimensionalen Computergrafik eingeführt. Die folgenden Kapitel beschreiben die Darstellung dreidimensionaler Objekte auf dem Bildschirm und welche Schritte dazu notwendig sind.

In der Computergrafik werden unter anderem drei Klassen von Darstellungsalgorithmen unterschieden: die klassische Darstellung („Rendering"), die Strahlverfolgung („Raytracing") und die Strahlungsverfahren („Radiosity"). Sie unterscheiden sich voneinander unter anderem extrem durch die Rechenzeiten, die erzeugte Bildqualität und die im einzelnen benötigten Algorithmen. Beispielsweise benötigt das Raytracing außer der Modellierungstransformation keine weiteren Transformationen. Ein Abschneiden der nicht sichtbaren Bereiche („Clipping") entfällt.

Allen Verfahren gemeinsam ist aber die Beleuchtung der darzustellenden Objekte. Im Grunde sind nämlich alle Algorithmen durch den Wunsch entstanden, Objekte möglichst realistisch darzustellen, wozu die „Simulation" von Beleuchtungsverhältnissen erheblich beiträgt. Im folgenden werden die drei genannten Klassen und ihre Gemeinsamkeiten beschrieben: Zuerst der klassische Ansatz, das Rendering. Da die einzelnen Teile dieses Verfahrens eine Folge von Teilalgorithmen beschreiben, spricht man auch von der „Rendering-Pipeline". Das Radiosity- und das Raytracing-Verfahren werden in größeren Abschnitten umfassend dargestellt.

Dieser Teil des Kapitels enthält auch den Einstieg in die Implementierung in Pascal, die bei Rendering und Radiosity verwendet werden. Gerade die Umsetzung in ein ablauffähiges Programm zeigt auf eine sehr einleuchtende Weise die Probleme, die dabei entstehen können.

7.1 Koordinatensysteme

Ein Hilfsmittel, um eine Objekt auf dem Bildschirm darzustellen, sind Transformationen. Sie „verwandeln" ein in einer dreidimensionalen Modellwelt beschriebenes Objekt in eine auf dem Bildschirm darstellbare Repräsentation. Dazu werden nacheinander verschiedene Koordinatensysteme verwendet, angefangen von Modellkoordinaten über Weltkoordinaten und ein normiertes Koordinatensy-

stem bis hin zu den Bildschirmkoordinaten. In jedem der Koordinatensysteme werden bestimmte Algorithmen verwendet, um die Darstellung eines Objektes zu berechnen.

Das Modell- und das Weltkoordinatensystem dienen vor allem dem „Zusammenbau" einer Szene, also der sogenannten Modellierung. Jedes, in seiner eigenen Umgebung entworfene Teil einer Szene kann mit Hilfe einer Transformation zwischen diesen Koordinatensystemen in der Szene lokalisiert werden. Dies soll an einem einfachen Beispiel verdeutlicht werden.

Der in Abb. I des farbigen Bildteils gezeigte Molekülring soll durch die mehrfach transformierte Darstellung einer Kugel erzeugt werden. Dazu gehen wir in zwei Schritten vor: Zuerst wird ein einzelnes Molekül aus drei verschieden großen Kugeln erzeugt. Anschließend wird dieses Molekül sechsmal in den Molekülring eingesetzt. Nach diesen Schritten liegt das Molekül also im Weltkoordinatensystem vor.

In diesem Beispiel haben wir noch nicht festgelegt, wie eine Kugel beschrieben wird. In der vom Programm verwendeten grafischen Eingabesprache (siehe Anhang A) kann die Kugel trotzdem verwendet werden, weil sie im System vordefiniert ist. Das folgende Programmstück zeigt das Molekül in der von uns definierten Eingabesprache GED. Ohne die Syntax und die Modellierungsart näher zu kennen, ist die Struktur der Modellierung erkennbar. Die Eingabesprache dient dazu, diese Modellierungsschritte in einer kompakten und lesbaren Form zu beschreiben.

```
GED_VERSION 1;
OBJEKT Atom; KUGEL FACET 20,10; ENDOBJEKT;

OBJEKT Molekuel;
  OE 20; VERSCHIEBE 2,0,0; ZEICHNE Atom;
  PUSH;
    DREHE Z,50; VERSCHIEBE 0.9,0,0; SKALIERE 0.7,0.7,0.7;
    OD 1,0,1; ZEICHNE Atom;
  POP;
  PUSH;
    DREHE Z,-50; VERSCHIEBE 0.9,0,0; SKALIERE 0.7,0.7,0.7;
    OD 0,1,1; ZEICHNE Atom;
  POP;
ENDOBJEKT;

GERICHTETELQ 1,1,1 1,5,-2;
AMBIENTELQ 0.5,0.5,0.5;
SZENE Molekuelring;
  OD 1,0,0; ZEICHNE Molekuel; DREHE Z,60;
  OD 0,1,0; ZEICHNE Molekuel; DREHE Z,60;
  OD 0,0,1; ZEICHNE Molekuel; DREHE Z,60;
  OD 1,1,0; ZEICHNE Molekuel; DREHE Z,60;
  OD 0,1,1; ZEICHNE Molekuel; DREHE Z,60;
  OD 1,0,1; ZEICHNE Molekuel;
END.
```

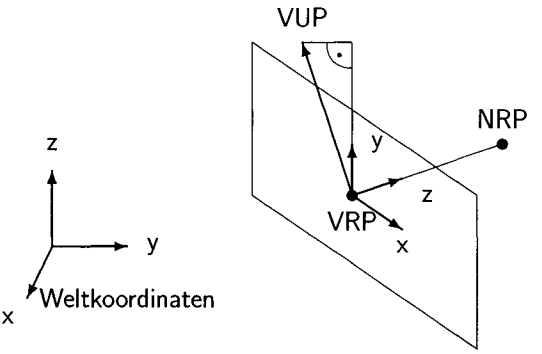


Abb. 7.1 Lage von Weltkoordinaten und Blickkoordinatensystem zueinander

Um vom Weltkoordinatensystem zu einem auf dem Bildschirm darstellbaren Objekt zu kommen, wird im nächsten Schritt die Festlegung der Sicht auf die Szene definiert. Die Festlegung der Blickrichtung, der Blickebene, der Projektionsart und der vorderen und hinteren Begrenzung des Sichtbereichs macht die Transformation von Objekten in das normierte Koordinatensystem möglich.

Da diese Transformationen relativ schwer zu beschreiben sind, sollen sie hier noch einmal kurz wiederholt werden. In homogenen Koordinaten $\begin{pmatrix} w & x & y & z \end{pmatrix}$ kann die Transformation eines Koordinatensystems A in ein Koordinatensystem B mit Hilfe der Bestimmung der Basis von B in A bestimmt werden. B_w ist die Koordinate des Ursprungs von B in A, B_x, B_y und B_z die Vektoren der Achsen. Dann ist die Transformation M_{B_A} durch

$$M_{B_A} = \begin{pmatrix} B_w \\ B_x \\ B_y \\ B_z \end{pmatrix}$$

definiert. M_{A_B} läßt sich als Inverse dieser Matrix berechnen. Im folgenden soll auf diese Weise die Transformation M_{WC_NPC} von Welt- in ein normiertes Koordinatensystem ausgedrückt werden.

Als Zwischenschritt wird ein weiteres Koordinatensystem, die Blickpunktkoordinaten (VRC $\hat{=}$ view reference coordinates), eingeführt. Wie Abb. 7.1 zu entnehmen ist, wird die Bildebene dadurch festgelegt.

Drei Größen bestimmen dieses System: Der Blickreferenzpunkt (VRP $\hat{=}$ view reference point), also der Punkt, auf den geschaut wird, legt den Ursprung des VRC-Systems in Weltkoordinaten fest. Der Normalenreferenzpunkt (NRP $\hat{=}$ normal reference point) bestimmt die Normale zur Bildebene. Der NRP liegt immer auf einer Geraden, die durch das Auge und den VRP geht. Im allgemeinen wird für ihn der Blickpunkt eingesetzt. Als dritte Größe schließlich legt der sogenannte „view up vector" (VUP) fest, wo auf dem Bildschirm „oben" liegen soll. Dieser Vektor bestimmt also die y-Richtung des VRC-Systems.

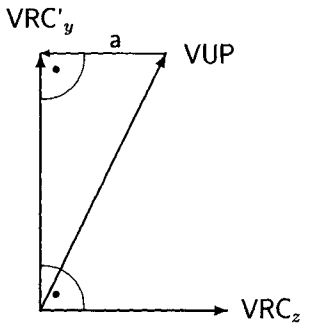


Abb. 7.2 Bestimmung des zweiten Basisvektors des Blickkoordinatensystems

Die Matrix M_{VRC_WC} setzt sich zeilenweise aus dem Ursprung und den Basisvektoren des VRC in Weltkoordinaten zusammen. Der Ursprung VRC_w entspricht dem Blickreferenzpunkt VRP. Am einfachsten zu berechnen ist der Basisvektor VRC_z. Dieser ist die Differenz von NRP und VRP. Er muß normiert werden, um die Länge 1 zu erhalten, also gilt:

$$VRC_z = \frac{NRP - VRP}{|NRP - VRP|}.$$

Der zweite Basisvektor VRC_y ergibt sich aus dem Basisvektor VRC_z und dem Vektor nach „oben" . Die Vektoren müssen senkrecht aufeinander stehen, daher soll

$$(VRC_y \cdot VRC_z) = 0$$

gelten. Weiterhin ergibt sich VRC'_y aus dem VUP, zu dem der Vektor $a \cdot VRC_z$ addiert wird, um genau diese Bedingung zu erfüllen (Abb. 7.2).

$$a = -(VUP \cdot VRC_z).$$

Also gilt:

$$VRC'_y = VUP - (VUP \cdot VRC_z)VRC_z,$$

und

$$VRC_y = \frac{VRC'_y}{|VRC'_y|}.$$

Der letzte Basisvektor VRC_x schließlich ist das Kreuzprodukt aus den bereits bekannten. daher

$$VRC_x = VRC_y \times VRC_z.$$

Die Transformationsmatrix M_{VRC_WC} ist also

$$M_{VRC_WC} = \begin{pmatrix} VRC_w \\ VRC_x \\ VRC_y \\ VRC_z \end{pmatrix}$$

M_{WC_VRC} ist die Inverse dieser Matrix.

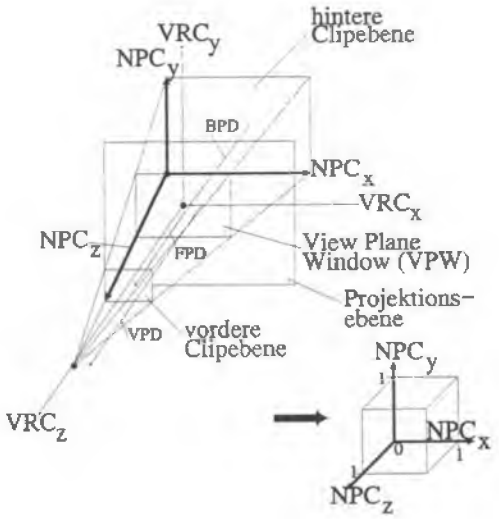


Abb. 7.3 Lage von Blickkoordinaten und normierten Projektionskoordinaten zueinander

Die Abbildung von Blickkoordinaten in das normierte Projektionskoordinatensystem ist nun relativ einfach zu berechnen (Abb. 7.3). Im normierten Koordinatensystem liegen alle Punkte im Einheitswürfel des ersten Oktanten. Die Abbildung wird durch den Projektionsreferenzpunkt PRP mit z-Koordinate VPD (view plane distance $\hat{=}$ Abstand zum Ursprung), das Projektionsfenster ($VPW \;\hat{=}$ view plane window) und die Abstände der vorderen und hinteren Sichtbegrenzung ($FPD, BPD \;\hat{=}$ front bzw. back plane distance) festgelegt.

Das Fenster ist durch $x_{min}, x_{max}, y_{min}$ und y_{max} festgelegt. Der Ursprung des NPC in Blickkoordinaten liegt bei

$$NPC_w = \left(\begin{array}{cccc} 1 & x_{min} \cdot \frac{VPD-BPD}{VPD} & y_{min} \cdot \frac{VPD-BPD}{VPD} & BPD \end{array} \right)$$

Dies kann leicht mit Hilfe des Strahlensatzes nachgewiesen werden. Analog ergeben sich die anderen Größen der Matrix. Die Resultierende ist:

$$M_{NPC_VRC} = \left(\begin{array}{cccc} 1 & k \cdot x_{min} & k \cdot y_{min} & BPD \\ 0 & k \cdot (x_{max} - x_{min}) & 0 & 0 \\ 0 & 0 & k \cdot (y_{max} - y_{min}) & 0 \\ l & 0 & 0 & l \cdot VPD \end{array} \right)$$

mit

$$k = \frac{VPD - BPD}{VPD}$$

und

$$l = \frac{FPD - BPD}{VPD - FPD}.$$

Die Inverse M_{VRC_NPC} wird nun mit der Matrix M_{WC_VRC} zur Gesamttransformation M_{WC_NPC} verknüpft.

$$\begin{aligned} M_{WC_NPC} &= M_{WC_VRC} \cdot M_{VRC_NPC} \\ &= (M_{NPC_VRC} \cdot M_{VRC_WC})^{-1}. \end{aligned}$$

Der Übergang von normierten Projektionskoordinaten in das Bildschirmkoordinatensystem entspricht nun nur noch einer Skalierung des normierten Bereiches auf Bildschirmgröße.

Statt einer perspektivischen kann natürlich auch eine Parallelprojektion zwischen dem Blickkoordinatensystem und dem normierten Projektionskoordinatensystem berechnet werden. Ein Projektionsreferenzpunkt ist dann nicht mehr notwendig, nur noch die Angabe von vorderer und hinterer Clipebene. Die Matrix M_{NPC_VRC} ist dann ohne Herleitung durch

$$M_{NPC_VRC} = \begin{pmatrix} 1 & x_{min} & y_{min} & BPD \\ 0 & x_{max} - x_{min} & 0 & 0 \\ 0 & 0 & y_{max} - y_{min} & 0 \\ 0 & 0 & 0 & FPD - BPD \end{pmatrix}$$

gegeben.

Tabelle 7.1 stellt nochmals die Abfolge der Koordinatensysteme innerhalb des Rendering zusammen.

Tabelle 7.1 Die Rendering-Pipeline mit den Entsprechungen zwischen den verschiedenen Koordinatensystemen und den zur Darstellung benutzten Algorithmen

Koordinatensystem	Algorithmus
Modellkoordinaten	lokale und hierarchische Modellierung
Weltkoordinaten	Plazierung der Objekte und Lichtquellen einer Szene, Berechnung der Beleuchtung, Festlegung der Sichtverhältnisse
Blickpunktkoordinaten	Hilfskoordinatensystem für die einfachere Berechnung der Sichtverhältnisse
Normierte Koordinaten	Entfernung der Rückflächen, Clipping (Abschneiden)
Bildschirmkoordinaten	Scan-Konvertierung der Objekte, Schattierungsrechnung, Verdeckungsrechnung

7.2 Modellierung

Während der Modellierung findet die Umsetzung der Idee dessen, was wir darstellen wollen, in ein mathematisches Modell statt. Objekte – die Dinge, die dargestellt werden sollen – nehmen einen bestimmten Raum ein. Diesen Raum zu beschreiben ist die Aufgabe der Modellierung.

Grundsätzlich wird ein Objekt auf verschiedene Art und Weise beschrieben:

1. Durch Aufzählung aller Punkte des Objektes. Soll z.B. eine Kugel beschrieben werden, so sind dies alle Punkte innerhalb eines bestimmten Radius um den Mittelpunkt.

$$\text{Kugel} \hat{=} \{P = \begin{pmatrix} x & y & z \end{pmatrix} \mid |P - M| \leq r\}$$

   Diese Beschreibung wird auch als *volumenorientierte* Modellierung bezeichnet.
2. Durch Beschreibung der „Grenzfläche" des Objektes. So wird beispielsweise die Oberfläche einer Kugel durch

$$\text{Kugel} \hat{=} \{P = \begin{pmatrix} x & y & z \end{pmatrix} \mid |P - M| = r\}$$

   angegeben. Diese Darstellungsform ist *flächenorientiert.*

Leider ist es nicht immer so einfach wie im Beispiel der Kugel, ein Objekt zu beschreiben. Im allgemeinen sind diese komplizierter aufgebaut. Teilweise können sie aber aus einfacheren Objekten zusammengefügt werden.

Das Zusammensetzen von volumenorientierten Primitiven (Kugel, Kegel, Zylinder, Torus, ...) mit Hilfe logischer Operationen (UND, ODER, NICHT) wird als CSG-Modellierung bezeichnet. CSG ist die Kurzform von „constructive solid geometry". Diese Form der Modellierung wird häufig im Zusammenhang mit dem Raytracing verwendet. Schnittpunktberechnungen mit den Grundkörpern können optimiert werden. Schnitte mit verknüpften Objekten werden auf die Verknüpfung der Schnitte zurückgeführt [Gla89].

Eine Verarbeitung von CSG-Modellen mit dem Rendering und dem Radiosity-Verfahren ist schwieriger, weil diese im wesentlichen Flächeninformationen für die Berechnungen benutzen. CSG-generierte Modelle müssen daher vor der Verarbeitung durch diese Algorithmen in eine flächenhafte Darstellung umgewandelt werden. Dabei geht im allgemeinen ein Teil der Zusammenhangsinformation verloren.

Oberflächen- bzw. flächenorientierte Darstellungen sind variabler einsetzbar und existieren in verschiedenen Formen. Generell werden exakte Beschreibungen und Annäherungen unterschieden. Eine exakte Information über die Oberfläche ist z.B. das oben angegebene Beispiel.

Neben dieser Aufzählungsform, die eine *implizite* Beschreibung der Oberfläche ist, existieren auch *explizite* Beschreibungen. Beispielsweise kann eine Ebene durch die Gleichung

$$z = f(x, y) = a \cdot x + b \cdot y + c$$

beschrieben werden. Eine weitere, wichtige Form der Repräsentation ist die Parameterdarstellung

$$P(u, v) = \begin{pmatrix} x(u, v) \\ y(u, v) \\ z(u, v) \end{pmatrix}.$$

Hier sind die Parameter u und v im allgemeinen in einem Bereich zwischen 0 und 1 frei wählbar.

Alle diese Beschreibungen sind exakte mathematische Modelle von Oberflächen. Oft liegt aber nur ein Teil der Oberfläche fest oder man möchte Objekte nur angenähert beschreiben, um Rechenzeit einzusparen. In diesem Fall werden im wesentlichen zwei Formen der Modellierung verwendet: Die Interpolation bzw. Approximation von gegebenen, festgelegten Punkten und die Annäherung durch ebene Flächenstücke.

Interpolationen und Approximationen sind die Erweiterung der Interpolation bzw. Approximation von Kurven im 2D. Durch Einführen einer dritten Komponente können Kurven im Raum beschrieben werden.

$$P(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix} \qquad 0 \leq t \leq 1$$

In den Grundlagenwerken zur Computergrafik (z.B. bei Foley et.al. [FvDFH90]) findet sich mehr zu diesem Thema. Auch diese Modelle werden bei der Darstellung oft auf die letzte Form zurückgeführt: die Approximation von Körpern und Flächen durch ebene Flächenstücke.

Es hat sich gezeigt, daß die Annäherung von Oberflächen durch ebene Flächenstücke viele Vorteile hat:

- Sie sind planar. Daher läßt sich z.B. der Normalenvektor im allgemeinen sehr einfach berechnen.
- Sie genügen einer linearen Ebenengleichung. Die Interpolation beliebiger Parameter, z.B. der Farbe über einer Ebene, ist ebenfalls linear.
- Sie sind einfach zu transformieren.
- Eine Verdeckungsrechnung ist wesentlich einfacher auszuführen als beispielsweise bei parametrisierten Flächen, bei denen unter anderem zuerst die Konturen bestimmt werden müssen.
- Die Umformung in das Rasterbild ist denkbar einfach.
- Schnitte zwischen Objekten oder Schnitte mit Strahlen sind Lösungen linearer Gleichungen.

Neben diesen Vorteilen stehen folgende Nachteile:

- Es treten Ecken auf, wo Rundungen erwartet werden. Diesem Effekt kann nur durch visuelles „Abrunden" oder mehr Ebenenstücke entgegengewirkt werden.
- Für komplexe Bilder wird eine große Anzahl der Ebenenstücke benötigt.

Zusammenfassend kann gesagt werden, daß diese Darstellungsform am häufigsten verwendet wird. Die für den Vorgang der Modellierung intuitiveren Modelle, z.B. die CSG-Modelle, werden für das Rendering umgewandelt – so auch in unserer Implementierung.

7.3 Entfernung der Rückflächen

Im normierten Koordinatensystem wird festgestellt, ob ein Objekt sichtbar oder vom Blickpunkt abgewendet ist. Durch dieses sogenannte „backface culling" (Entfernen der rückwärtigen Flächen) kann der Berechnungsaufwand für die Darstellung fast halbiert werden, wenn er frühzeitig stattfindet.

Eine Ebene ist dann von einem Punkt P_E aus sichtbar, wenn für die Normale $\mathbf{N}$ der Ebene

$$(\mathbf{N} \cdot P_E) = d > 0$$

gilt. Ist $d = 0$, so liegt der Punkt auf der Ebene. Für $d < 0$ liegt er auf der Rückseite.

Ist also das Skalarprodukt aus dem Blickvektor von P_E zum Blickpunkt E und der Normale $\mathbf{N}$ einer Fläche kleiner als 0, so braucht die Fläche bei der Darstellung nicht berücksichtigt zu werden. Damit ist viel Rechenarbeit gespart.

Die Bedingung für die Entfernung einer Fläche läßt sich in kartesischen Weltkoordinaten einfach umformulieren zu:

$$(\mathbf{N} \cdot (E - P_E)) - d = (\mathbf{N} \cdot P') - d < 0,$$

was gleichbedeutend ist mit:

$$\begin{aligned}
\begin{pmatrix} n_x & n_y & n_z \end{pmatrix} \begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} - d &< 0 \\
\Longleftrightarrow n_x p'_x + n_y p'_y + n_z p'_z - d \cdot 1 &< 0 \\
\Longleftrightarrow \begin{pmatrix} -d & n_x & n_y & n_z \end{pmatrix} \begin{pmatrix} 1 \\ p'_x \\ p'_y \\ p'_z \end{pmatrix} &< 0 \\
\Longleftrightarrow (\tilde{N} \cdot \tilde{E}) &< 0
\end{aligned}$$

in homogenen Weltkoordinaten. Diese Ungleichung kann in normierte homogene Koordinaten transformiert werden. In diesem Koordinatensystem ist die Ungleichung besonders einfach zu testen, da E_{WC} definitionsgemäß auf den Vektor $E_{NPC} = \begin{pmatrix} 0 & 0 & 0 & 1 \end{pmatrix}$ abgebildet wird. Wenn M_{WC_NPC} die Matrix zur Transformation von homogenen Punkten ist, dann wird der Normalenvektor mit

$$((M_{WC_NPC})^{-1})^T = (M_{NPC_WC})^T$$

transformiert. Daher ergibt sich für den Test:

$$\begin{aligned}
(\tilde{N} \cdot \tilde{E}) &< 0 \\
\Longrightarrow (\tilde{N} M^T_{NPC_WC} \cdot \tilde{E} M_{WC_NPC}) &< 0 \\
\Longleftrightarrow (\tilde{N} M^T_{NPC_WC} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}) &< 0
\end{aligned}$$

$$\Longleftrightarrow (\tilde{N} M_{NPC_WC}^{T})_z \;<\; 0.$$

Es wird also nur eine Zeile der Matrixberechnung auch tatsächlich ausgeführt, um eine Fläche als Vorder- oder Rückfläche zu klassifizieren.

7.4 Clipping

Das Clipping, also das Abschneiden der nicht sichtbaren Teile, kann einfach auf drei Dimensionen erweitert werden. Der sichtbare Bereich ist nun ein Würfel, im normierten Koordinatensystem NPC der Einheitswürfel im ersten Oktanten. Wir unterscheiden auch hier das Clippen von Linien und Polygonen.

Die Erweiterung des Algorithmus von Cohen und Sutherland betrachtet nun nicht vier Begrenzungsgeraden, sondern sechs Begrenzungsebenen. Bezeichnen wir sie als x_{min}, x_{max}, y_{min}, y_{max}, z_{min} und z_{max}. Für jeden Endpunkt einer Linie werden nun sechs Bit mit den logischen Aussagen

$$\begin{aligned} x &< x_{min} \\ x &> x_{max} \\ y &< y_{min} \\ y &> y_{max} \\ z &< z_{min} \qquad \text{und} \\ z &> z_{max} \end{aligned}$$

bestimmt und zugeordnet.

Eine Linie liegt genau dann außerhalb des sichtbaren Bereiches, wenn die UND-Verknüpfung der beiden Bitfelder nicht Null ist. Sie liegt dagegen vollständig innerhalb des Sichtfeldes, wenn die Codes beider Endpunkte Null ist. In allen anderen Fällen muß die Linie weiter untersucht bzw. die Verkürzung berechnet werden.

Dieser Algorithmus kann für die einfache und schnelle Akzeptanz oder Ablehnung eines Polygons erweitert werden. Es gilt analog zur Linie: Ist der Code *aller* Eckpunkte des Polygons Null, so liegt es vollständig innerhalb des sichtbaren Bereiches. Ist die UND-Verknüpfung aller Eckpunkt-Beurteilungen nicht Null, so liegt es vollständig außerhalb. Ganz analog wird der Polygon-Clipping-Algorithmus von Sutherland und Hodgman auf drei Dimensionen erweitert.

Eine Verkomplizierung der Algorithmen ergibt sich durch die Tatsache, daß in homogenen Koordinaten gerechnet wird. Dazu soll das Clippen am Einheitswürfel des normierten Projektionskoordinatensystems betrachtet werden. Für x', y' und z' eines Punktes $\begin{pmatrix} x' & y' & z' \end{pmatrix}$ soll

$$0 \leq x' \leq 1, \qquad 0 \leq y' \leq 1 \qquad \text{und} \qquad 0 \leq z' \leq 1$$

gelten. Nach einer homogenen Erweiterung entspricht dies aber

$$P = \begin{pmatrix} w & x & y & z \end{pmatrix}$$

mit:

$$0 \leq x/w \leq 1, \qquad 0 \leq y/w \leq 1 \qquad \text{und} \qquad 0 \leq z/w \leq 1.$$

Also wird für homogen beschriebene Punkte der Bereich $0 \leq x' \leq 1$ durch die Abfrage

$$0 \leq x \leq w$$

getestet. Die Anwendung dieser Überlegungen auf andere Sichtbereiche verläuft analog.

7.5 Verdeckungsrechnung

Das normierte Koordinatensystem dient dazu, den sichtbaren Bereich eines Bildes festzulegen. Alle sich innerhalb einer Szene befindenden Objekte werden deshalb an diesem Bereich abgeschnitten. Nach diesem Schritt steht prinzipiell fest, welche Objekte in einem Bild sichtbar sind und welche nicht. Die darauf folgenden Algorithmen gehen nicht mehr darauf ein, ob ein Objekt dargestellt wird oder nicht. Es stellt sich jetzt die Frage, *was* von einem Objekt dargestellt wird, um das Bild realistisch zu gestalten.

Ein Schritt, der wesentlich dazu beiträgt, daß eine Darstellung realistisch wirkt, ist die Verdeckungsrechnung. Sie sorgt dafür, daß vorne liegende Objekte erscheinen und verdeckt liegende Flächen nicht dargestellt werden. Die Verdeckungsrechnung basiert auf den Bildschirmkoordinaten bzw. normierten Projektionskoordinaten.

Prinzipiell werden bei der Verdeckungsrechnung zwei Ansätze unterschieden: der pixelorientierte und der objektorientierte [FvDFH90]. Im ersten Fall wird jedes der p Pixel eines Bildschirms untersucht. Alle Objekte werden bezüglich der Sichtbarkeit getestet und die Pixelfarbe entsprechend gesetzt.

Algorithmus 7.1 (Pixelorientierte Verdeckung)

FÜR jedes der p Pixel des Bildschirms:
Bestimme das Objekt, welches am nächsten zum ..
.. Betrachtungsstandpunkt liegt.
Setze die Pixelfarbe entsprechend.
ENDE FÜR jedes Pixel.

Gibt es n Objekte in der Szene, so ist der Aufwand für diesen Algorithmus proportional zu $n \cdot p$. Das Raytracing zur Bestimmung der verdeckten Flächen gehört in diese Kategorie der Verdeckungsalgorithmen.

Der objektorientierte Ansatz vergleicht jedes Objekt mit allen anderen Objekten der Szene und entfernt dabei die nicht sichtbaren Teile.

Algorithmus 7.2 (Objektorientierte Verdeckung)

FÜR jedes der n Objekte der Szene:
Bestimme die Teile des Objektes, die nicht von sich selbst ..
.. oder anderen Objekten verdeckt sind.
Stelle diese Teile in der entsprechenden Farbe dar.
ENDE FÜR für jedes Objekt.

Der Aufwand liegt hier in der Größenordnung n^2.

Diese generelle Klassifikation vernachläßigt allerdings Aspekte der Vereinfachung der Algorithmen durch Berücksichtigung von Zusammenhängen. In einem klassischen Artikel haben Sutherland, Sproull und Schumacker 1974 diese Zusammenhänge untersucht und klassifiziert [SSS74]. Wir stellen im folgenden einige dieser Algorithmen vor.

7.5.1 Der Maler-Algorithmus

Die einfachste Art und Weise, das Problem der Verdeckung zu lösen, ist die Darstellung aller Objekte von hinten nach vorne. Dabei werden die verdeckten Flächen automatisch verdeckt, die sichtbaren bleiben erhalten. Voraussetzung für die Machbarkeit des Algorithmus ist allerdings, daß für je zwei Objekte eindeutig definiert sein muß, welches der beiden „hinten" und welches „vorne" liegt. Schon bei einfachen Szenen, z.B. der Darstellung eines Würfels, kann dies nicht berechnet werden. Die Seitenwände des Würfels beispielsweise werden i.a. nicht voneinander verdeckt, aber ein „vorne" oder „hinten" kann nicht angegeben werden. Aus diesen Gründen wird der Maler-Algorithmus heute meist nur noch im 2D-Bereich eingesetzt.

7.5.2 Der Tiefensortier-Algorithmus

Dieser Algorithmus ist eine Erweiterung des Maler-Algorithmus. Gibt es Zweideutigkeiten in der Ordnung des z-Wertes zweier Polygone, so werden diese so lange unterteilt, bis die Eindeutigkeit hergestellt ist. Dazu sind eine Menge von Tests der beiden Objekte zueinander notwendig. Sie beziehen unter anderem die umhüllenden Quader (bounding box) für die Tests mit ein. Es können allerdings nicht alle denkbaren Konstellationen mit diesem Algorithmus behandelt werden, insbesondere bei sich durchdringenden Objekten gibt es Probleme.

7.5.3 Der z-Buffer-Algorithmus

Der z-Buffer ist ein schneller, aber speicherplatzintensiver Mechanismus zur Berechnung der Verdeckung. Er basiert auf der gleichzeitigen Speicherung von

Farb- und Tiefenwerten (z-Komponente) in einem Speicher. Seine genaue Beschreibung erfolgt in Abschnitt 13.1.

7.5.4 Scanline-Algorithmen

Diese Klasse von Algorithmen arbeitet die Objekte in sortierter Reihenfolge ab. Eine erste Sortierung ordnet Objekte in aufsteigender y-Folge an. In einem zweiten Sortierschritt werden für jeden konstanten y-Wert die x-Werte, an denen Polygone die Scanline schneiden, sortiert. Das Resultat ist eine Liste von Scanlinienabschnitten, sogenannten „spans", die dann überlappungsfrei dargestellt werden können.

Es existiert eine Vielzahl von Verdeckungsalgorithmen, die hier verständlicherweise nicht alle dargestellt werden können. Da Speicherplatz inzwischen auch in Größenordnungen, die für den z-Buffer benötigt werden, preisgünstig sind, hat sich dieser Algorithmus doch weitestgehend durchgesetzt.

7.6 Beleuchtungs- und Schattierungsverfahren

Alle Verfahren, die sich damit beschäftigen, welche Farbe ein Objekt hat, sind unter den Begriffen Beleuchtungs- und Schattierungsrechnung zusammengefaßt. Die Beleuchtungsrechnung bestimmt die Farbe eines Objektes in Weltkoordinaten unter Berücksichtigung physikalischer Gesetzmäßigkeiten. Die Schattierungsrechnung dagegen bestimmt die auf dem Bildschirm sichtbare Farbe. Unter anderem wird dabei die berechnete Farbe des Objektes in eine der wählbaren Farben auf dem Bildschirm umgerechnet. Diese Verfahren werden in den folgenden Abschnitten noch genauer vorgestellt.

7.7 Scankonvertierung

Der letzte Schritt bei der Darstellung von Objekten ist die Umwandlung der kontinuierlich gedachten Modelle in das auf dem Bildschirm darstellbare Raster von farbigen Bildpunkten (Pixel). Dieser Schritt heißt Scankonvertierung. Linien und Flächen wurden bislang als Punkte bzw. Sammlungen von Punkten behandelt. Beim Übergang auf die heute übliche Rasterdarstellung des Bildschirms mit einer Auswahl von Farben müssen wir diese Darstellung verlassen. Nun werden Pixel mit deren Adresse und einem Farbwert benötigt.

Die grundlegenden Ausgabeprimitive sind Linien und Polygone. Für beide gibt es auch in der Scankonvertierung die verschiedensten Methoden und Algorithmen. Zwei davon werden nun vorgestellt.

Die einfachste Art, Linien auf einem Rastergerät darzustellen, besteht darin, ausgehend vom Startpunkt der Linie die x- und y-Werte gemäß der Steigung zu erhöhen. Für die Darstellung wird der erhaltene Wert auf den nächsten dar-

stellbaren Punkt des Bildschirms abgebildet. Dieser Algorithmus nennt sich (unsymmetrischer) DDA (digital differential analyzer).

Es sei der Anfangspunkt $\begin{pmatrix} x_a & y_a \end{pmatrix}$ und der Endpunkt $\begin{pmatrix} x_e & y_e \end{pmatrix}$ einer Strecke mit $x_a < x_e$ gegeben. Die Darstellung der Geraden ist dann

$$y = y_a + m \cdot (x - x_a) \qquad \text{mit} \qquad m = \frac{y_e - y_a}{x_e - x_a}.$$

Der Algorithmus lautet dann für Strecken mit Steigungen zwischen 0 und 45 Grad:

```
dx := xe - xa;
dy := ye - ya;
y := ya + 0.5;          { Anfangswert+0.5 aus Symmetriegruenden }
dxy := dy/dx;           { Steigung }
x := ROUND(xa);         { Anfangswert }
Plot(x,int(y));         { Anfangspunkt }
WHILE x < xe DO BEGIN
   INC(x);
   y := y + dxy;
   Plot(x,ROUND(y))
END;
```

Der Sonderfall $x_e = x_a$ muß abgefangen werden.

Weitere bekannte Algorithmen wie z.B. der Bresenham-Algorithmus, der nur ganze Zahlen benutzt, können in bestimmten Fällen noch effizienter sein.

Die Darstellung von Flächen kann auf die Darstellung von Linien zurückgeführt werden, indem für jede Rasterlinie alle Abschnitte (spans) bestimmt werden, die eine Fläche belegt. Der folgende Algorithmus bestimmt für ein gegebenes Polygon die Schnittpunkte mit der jeweiligen Rasterzeile und sammelt sie in einer Liste. Nach dem Sammeln werden diese Zeilenabschnitte in der entsprechenden Farbe dargestellt.

Algorithmus 7.3 (Flächenfüllen)

FÜR alle Rasterzeilen y:
  Erzeuge eine leere Liste.
  FÜR alle Kanten des Polygons:
    WENN die Rasterzeile und die Kante sich in einem..
    ..Punkt P schneiden:
      Trage den Punkt P in die Liste ein.
    ENDE WENN die Rasterzeile
  ENDE FÜR alle Kanten.
  FÜR alle Spans in der Liste:
    Fülle den Zeilenabschnitt.
  ENDE FÜR alle Spans.
ENDE FÜR alle Rasterzeilen.

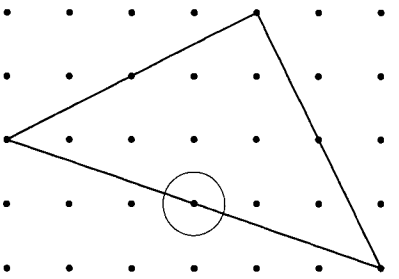


Abb. 7.4 Abtastung eines Dreiecks durch ein Raster

7.8 Antialiasing

Aufgrund der Abtastung durch das Raster des Bildschirms entstehen Fehler, die als sogenannte Alias-Effekte sichtbar werden. Dieser Effekt wird durch Abtastung der Zeit in Animationen noch verstärkt. Die beiden Effekte werden als räumliches bzw. zeitliches Aliasing bezeichnet. Gegenmaßnahmen werden entsprechend „Antialiasing" genannt. Hier wird nur das räumliche Abtasten betrachtet.

Das Problem des Aliasing tritt durch die Darstellung nur bestimmter Punkte in einem gerasterten Bild auf (Abb. 7.4). Korrekt wäre eigentlich eine Integration der Intensität (d.h. der Farbe), die unter der Fläche eines Pixels liegen. Eine alte Streitfrage ist übrigens, ob dafür die Pixelfläche als rund oder als eckig anzunehmen sei. Die Effekte der Unterabtastung äußern sich folgendermaßen:

- Kanten erscheinen treppenförmig.
- Lange, schmale Linien werden eventuell unterbrochen dargestellt („Perlenkette").
- Kleine Veränderungen der Objektposition rufen scheinbare Veränderungen der Objektform hervor.
- Kleine Objekte „blinken" bei Bewegung, je nachdem, ob sie den Pixelmittelpunkt treffen oder nicht.

Lösungsansätze für das Antialiasing sind die folgenden Verfahren:

- Alle Intensitätsanteile an einer Pixelfläche werden korrekt integriert. Diese exakte Lösung ist schon deshalb oft nicht durchführbar, weil Berandungen nicht unbedingt analytisch beschreibbar sind. Zusätzlich treten Probleme auf, wenn sich die Intensität eines Objektes innerhalb des Pixels ändert wie bei den üblichen Schattierungsalgorithmen.
- Die Integration wird durch eine numerische Näherung, die Summation, ersetzt. Dies entspricht einer Überabtastung von sogenannten Subpixeln

innerhalb des Pixels. Bereits mit 4×4 bis 8×8 Subpixeln werden visuell ansprechende Bilder erzeugt. Das Verfahren kann zudem auf diejenigen Pixel beschränkt werden, für die es notwendig ist, also beim Auftreten von Kanten. Da der Aufwand des Algorithmus quadratisch mit der Zahl der zusätzlichen Abtastungen wächst, ist er allerdings sehr zeitintensiv.

- Die regelmäßigen Überabtastwerte des letzten Verfahrens werden durch unregelmäßige ersetzt. Es werden geringere Überabtastungen benötigt, da durch die statistische Verteilung ein vom Auge weniger wahrgenommenes „Rauschen" über das Bild gelegt wird.
- Zusätzlich kann die als gleichmäßig angenommene Gewichtung aus dem zweitgenannten Verfahren durch andere Filterformen ersetzt werden. Dies erzeugt eine zusätzliche Tiefpaßwirkung im endültigen Bild. Üblich sind Dreiecksfilter, Gauß-Filter und sinc-Filter.

8. Beispiele zu den Grundlagen dreidimensionaler Computergrafik

In diesem Kapitel werden die Grundlagen dreidimensionaler Computergrafik anhand von Beispielen wiederholt. Dazu wollen wir zuerst das Beispiel des Molekülrings nochmals aufgreifen. Es soll die Modellierung, die verschiedenen Transformationen, das Clipping, die Verdeckungsrechnung und die Scankonvertierung verdeutlichen.

8.1 Modellierung und Transformation zwischen Koordinatensystemen

Im Kap. 6 war mit einem Würfel eine einfache Szenerie aufgebaut worden. Ebenso wie `WUERFEL` ist `KUGEL` ein vordefiniertes Objekt. Weitere solcher Objekte sind `ZYLINDER`, `KEGEL` und `TORUS`. Sie sind in ihren Ausdehnungen jeweils in Bezug auf ein bestimmtes Koordinatensystem definiert. Dieses wird als Modellkoordinatensystem bezeichnet.

Da diese vordefinierten Körper aus Polygonen zusammengesetzt sind, ist optional mit dem Schlüsselwort `FACET` die Feinheit der Unterteilung der Körper einstellbar. Experimentieren Sie beispielsweise mit der Unterteilung der Kugel.

```
KUGEL;
```

Mit dieser `GED`-Zeile erzeugen Sie die Standardunterteilung, nämlich 16 Flächen entlang des „Äquators" und 8 Flächen von Pol zu Pol der Kugel (siehe Tabelle A.2). Sollte die Kugel auf Ihrem Bildschirm nicht rund erscheinen, dann kann mit der Angabe `PIXELRATIO` das Verhältnis zwischen Länge und Breite eines Pixels verstellt werden.

```
KUGEL FACET 4,2;
```

Diese Unterteilung erzeugt 4 Facetten in horizontaler und 2 Facetten in vertikaler Richtung: die Kugel ist zu einem Oktaeder degeneriert. Mit Hilfe der Option

```
(W)ireframe darstellen
```

im Menu des Programms können Sie die Facettierung sehr gut beobachten.

Um aus mehreren Kugeln das Kalottenmodell eines Moleküls zusammenzusetzen, können die Kugeln mit Hilfe von Transformationen an die richtige Stelle plaziert werden. Im Modellkoordinatensystem liegt der Mittelpunkt der Kugel im Ursprung, der Radius der Kugel beträgt 1. Ein Benzolring mit 6 Atomen kann z.B. durch die GED-Beschreibung

```
GED_VERSION 1;

GERICHTETELQ 1,1,1 1,5,-2;
SZENE Beispiel2;
                                    VERSCHIEBE 2,0,0; KUGEL;
   VERSCHIEBE -2,0,0; DREHE Z,60; VERSCHIEBE 2,0,0; KUGEL;
   VERSCHIEBE -2,0,0; DREHE Z,60; VERSCHIEBE 2,0,0; KUGEL;
   VERSCHIEBE -2,0,0; DREHE Z,60; VERSCHIEBE 2,0,0; KUGEL;
   VERSCHIEBE -2,0,0; DREHE Z,60; VERSCHIEBE 2,0,0; KUGEL;
   VERSCHIEBE -2,0,0; DREHE Z,60; VERSCHIEBE 2,0,0; KUGEL;
END.
```

generiert werden.

Warum muß dieses Beispiel so umständlich formuliert werden? Ein Grund dafür ist die fortlaufende Verknüpfung der Transformationen. Für jeden der sechs KUGEL-Aufrufe werden *alle Transformationen akkumuliert*, die jeweils vor diesem Befehl stehen. Damit ergibt sich für die erste Kugel die Folge:

```
VERSCHIEBE 2,0,0; KUGEL;
```

für die zweite die Folge:

```
VERSCHIEBE  2,0,0;
VERSCHIEBE -2,0,0;
DREHE Z,60;
VERSCHIEBE  2,0,0; KUGEL;
```

oder kürzer geschrieben:

```
DREHE Z,60;
VERSCHIEBE 2,0,0; KUGEL;
```

da sich die beiden Translationen gegenseitig neutralisieren, usw.

Der zweite Grund für die Formulierung des Beispiels ist die Reihenfolge der Ausführung der Transformationen. Im Prinzip besteht bei der Verkettung von Operationen

```
M1;
M2;
OBJEKT;
```

die Möglichkeit, zuerst M1 und dann M2 auf das Objekt anzuwenden, um dessen aktuelle Position und Größe zu bestimmen. Wir haben uns für die zweite Interpretation entschieden, d.h.

Transformationen werden in der umgekehrten Reihenfolge des Programmtextes auf Objekte angewendet.

Diese Festlegung hat Vorteile bei der „Prozeduralisierung" von Teilobjekten, wie Sie gleich sehen werden.

In unserem Beispiel wird also für die zweite Kugel zuerst die Translation um $\begin{pmatrix} 2 & 0 & 0 \end{pmatrix}$ ausgeführt und anschließend eine Drehung um die z-Achse um 60°. Dennoch ist es natürlich ziemlich umständlich, aufgrund der fortlaufenden Verkettung Operationen zuerst einzuführen und anschließend wieder rückgängig zu machen. Mit den beiden Befehlen `PUSH` und `POP` können aktuelle Transformations„zustände" gespeichert (aber nicht gelöscht) und wiederhergestellt werden.

Das Beispiel kann damit umgeschrieben werden:

```
GED_VERSION 1;

GERICHTETELQ 1,1,1 1,5,-2;
SZENE Beispiel3;
  PUSH; VERSCHIEBE 2,0,0; KUGEL; POP;
  DREHE Z,60;
  PUSH; VERSCHIEBE 2,0,0; KUGEL; POP;
  DREHE Z,60;
  PUSH; VERSCHIEBE 2,0,0; KUGEL; POP;
  DREHE Z,60;
  PUSH; VERSCHIEBE 2,0,0; KUGEL; POP;
  DREHE Z,60;
  PUSH; VERSCHIEBE 2,0,0; KUGEL; POP;
  DREHE Z,60;
  PUSH; VERSCHIEBE 2,0,0; KUGEL; POP;
END.
```

Das letzte `PUSH-POP`-Paar ist nicht unbedingt notwendig, dient uns aber im folgenden bei der weiteren Zusammenfassung. In solche Paare gefaßte Befehle der Eingabesprache können als „Prozeduren", Objekte genannt, vereinfacht werden.

Die Definition des Objektes in unserem Beispiel lautet dann:

```
OBJEKT Atom;
  VERSCHIEBE 2,0,0;
  KUGEL;
ENDOBJEKT;
```

Es wird mit

```
ZEICHNE Atom;
```

aufgerufen. Das `PUSH-POP`-Paar ist in diesem Aufruf implizit enthalten. Das umformulierte Beispiel ist nun also noch kompakter geworden.

```
GED_VERSION 1;

OBJEKT Atom;
  VERSCHIEBE 2,0,0;
  KUGEL;
ENDOBJEKT;

GERICHTETELQ 1,1,1 1,5,-2;
SZENE Beispiel4;
                ZEICHNE Atom;
  DREHE Z,60; ZEICHNE Atom;
  DREHE Z,60; ZEICHNE Atom;
  DREHE Z,60; ZEICHNE Atom;
  DREHE Z,60; ZEICHNE Atom;
  DREHE Z,60; ZEICHNE Atom;
END.
```

Objektdefinitionen stehen vor den Definitionen der Lichtquellen in der GED-Datei. Innerhalb von Objekten können auch wieder Objekte aufgerufen werden, so daß Sie z.B. ein Atom selbst durch ein komplexeres Molekül ersetzen können, etc. So kommen Sie dann zu der in Kap. 7 dargestellten GED-Beschreibung des Moleküls in Abb. I.

8.2 Weitere Koordinatensysteme

Neben Modell- und Weltkoordinaten, die implizit durch die Objekthierarchie gegeben sind, ist in PRIM auch die Beeinflussung der Blickpunkt-, der Bildschirm- und der normierten Projektionskoordinaten möglich. Sie werden durch die Betrachtungsangaben beeinflußt, die direkt vor den Lichtquellen stehen. Die Berücksichtigung der Angaben in den Transformationsmatrizen können Sie Kap. 7 entnehmen, die Syntax Abschnitt A.7.

Die Blickpunktkoordinaten werden durch AUGE, REFERENZPUNKT und UP-VEKTOR angegeben, die normierten Projektionskoordinaten durch BILDABSTAND, BILDBREITE, VORDERCLIPEBENE und HINTERCLIPEBENE festgelegt. Die Bildschirmkoordinaten sind mit BILDSCHIRMFENSTER und PIXELRATIO verstellbar. Ein Teil dieser Parameter kann auch interaktiv während des Programmlaufs verändert werden. Experimentieren Sie mit dem Molekül, indem Sie beispielsweise PIXELRATIO einmal auf den Wert 2.0 und ein anderes Mal auf den Wert 0.5 setzen.

8.3 Clipping

Das Wegschneiden von Bildteilen können Sie interaktiv sehr gut beobachten, indem Sie die vordere Clipebene mit Hilfe des Menueintrages von PRIM verschieben. Der voreingestellte Wert ist 10.0. Setzen Sie ihn auf 2.5 und stellen

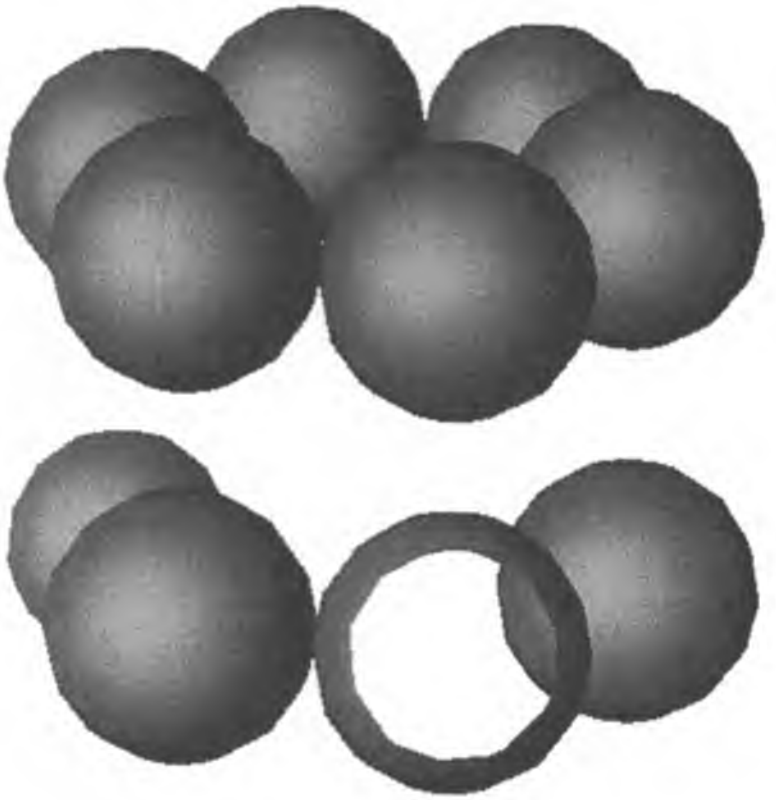


Abb. 8.1 Das Molekül vor (oben) und nach (unten) dem Clipping an der vorderen und hinteren Clipebene

das Bild anschließend neu dar, so ist nur noch ein Teil des Moleküls sichtbar. Stellen Sie auch noch die hintere Clipebene kleiner ein, z.B. auf den Wert 0.0, dann sehen Sie nur noch eine „Scheibe", die aus dem Molekül herausgeschnitten wird. Auch die entfernten Rückflächen der Kugeln können hiermit sehr gut beobachtet werden (Abb. 8.1).

9. Implementierung der grundlegenden Algorithmen

Im folgenden stellen wir diejenigen Module von PRIM vor, welche die im letzten Abschnitt besprochenen Inhalte in der Implementierung widerspiegeln. Details dazu finden sich auch im Anhang des Buches: „Programmierung von Rendering und Radiosity".

9.1 Die von PRIM verwendeten Units

Durch das Unit-Konzept von Turbo-Pascal läßt sich eine Modularisierung der Software erreichen, ohne die ein fast 9000 Zeilen umfassendes Programm unübersichtlich wäre.

Das gesamte Programmpaket umfaßt diverse Module, die grob in folgende Bereiche aufteilbar sind: Hilfs-Units und Units mit einer spezialisierten Aufgabe.

Zu den Hilfs-Units gehört PRIMGLOB, in der alle global verwendeten Typen und Konstanten definiert sind. PRIMGLOB wird von den meisten anderen Units schon im Interface-Teil verwendet.

Die zweite Hilfs-Unit ist PRIMOUT. Sie wird wegen ihrer Fehlerausgaberoutine in fast jedem Modul verwendet. Die anderen Routinen von PRIMOUT werden lediglich von wenigen Units benutzt.

Die dritte und letzte Hilfs-Unit ist PRIMMEM. Nur mit ihrer Hilfe ist es den anderen Modulen möglich, auf globale Punkte und Flächen, sowie auf z-, Item- und Textur-Buffer zuzugreifen. Die Aufgabe der einzelnen Module und welche anderen Units sie verwenden, geht aus Tabelle 9.1 hervor.

9.2 Die Unit PRIMVIEW, Matrizenrechnung zum Anfassen

Im Modul PRIMVIEW sind alle Routinen zusammengefaßt, die Matrizen *generieren*, sowie einige Hilfsroutinen implementiert, die im vorigen Abschnitt erklärt wurden. Zur ersten Gruppe gehören die folgenden Routinen:

```
PROCEDURE MultipliziereMatrix(a, b: Mat4Typ; VAR c: Mat4TYP);
PROCEDURE MultipliziereMatrix1(a: Mat4TYP; VAR c: Mat4TYP);
PROCEDURE InvertiereMatrix(a: Mat4TYP; VAR am1: Mat4TYP);
```

Tabelle 9.1 Aufgaben und Beziehungen der Module

Name der Unit	Beschreibung	Benutzte Units: (1) Im Interfaceteil (2) Im Implementationsteil
PRIM	Hauptprogramm	(1) – (2) crt, PrimGlob, PrimRead, PrimDere, PrimShow, PrimOut, PrimMem, PrimRend, PrimRad, PrimView
PRIMGLOB	globale Typen und Konstanten	(1) – (2) –
PRIMREAD	Einlesen einer GED-Datei	(1) PrimGlob (2) crt, PrimView, PrimOut, PrimMem
PRIMDEREF	Umwandlung in globale Punkt- und Flächenliste	(1) PrimGlob (2) crt, PrimOut, PrimView, PrimMem
PRIMSHOW	Darstellen der Szene als Drahtmodell	(1) PrimGlob (2) crt, PrimOut, PrimMem, PrimView
PRIMOUT	Fehlerausgabe, Pixel-Routinen	(1) – (2) dos, crt
PRIMMEM	Speicherverwaltungsroutinen	(1) PrimGlob (2) dos, PrimOut
PRIMREND	Rendering	(1) PrimGlob (2) PrimOut, PrimView, PrimLight, PrimShd, PrimMem, PrimRad
PRIMRAD	Radiosity-Prozeduren	(1) PrimGlob (2) crt, PrimView, PrimMem, PrimOut
PRIMVIEW	Transformationen	(1) PrimGlob (2) PrimOut
PRIMLIGHT	Beleuchtung	(1) PrimGlob (2) PrimView, PrimOut
PRIMSHD	Schattieren Texturieren	(1) PrimGlob (2) PrimOut, PrimLight, PrimRad, PrimView, PrimClip, PrimMem, PrimColt.INC (Farbtabelle für Marmor)
PRIMCLIP	Clipping	(1) PrimGlob (2) PrimShd, PrimView

```
PROCEDURE SetzeBetrachter(Auge, ReferenzPunkt: Punkt3DTYP;
                          UPVektor: Vektor3DTYP;
                          xMin, yMin, xMax, yMax,
                          BildAbstand,
                          VorderClipEbene, HinterClipEbene: SINGLE;
                          VAR WC_NPC, NPC_WC: MAT4TYP);
PROCEDURE SetzeBildschirm(x, y, Breite, Hoehe: INTEGER;
                          VAR NPC_DC: MAT4TYP);
```

Die implementierten Hilfsroutinen sind:

```
PROCEDURE NormiereVektor(VAR v: Vektor3DTYP);
FUNCTION IstRueckflaeche(n, p: Vektor3DTYP;
                         VAR aktNPC_WC: Mat4TYP): BOOLEAN;
PROCEDURE WCnachNPCUndClipCode(p3: Vektor3DTYP;
                               VAR p4: Vektor4DTYP;
                               VAR aktWC_NPC: Mat4TYP;
                               VAR ClipCode: BYTE);
PROCEDURE NPCnachDC(pNPC: Vektor4DTYP; VAR aktNPC_DC: Mat4TYP;
                    VAR pDC: Vektor3DTYP);
```

`MultipliziereMatrix` multipliziert die 4×4-Matrizen `a` und `b` und liefert als Ergebnis die Matrix `c`. `MultipliziereMatrix1` multiplizert `a` mit `c` und liefert in `c` das Ergebnis der Multiplikation zurück. `MultipliziereMatrix1` dient im wesentlichen zur Verkettung von Transformationen.

Mit Hilfe von `InvertiereMatrix` kann eine Abbildung in ihre Umkehrabbildung umgerechnet werden. `SetzeBetrachter` bestimmt die Abbildungsmatrix `WC_NPC` und ihre Inverse `NPC_WC`. Dabei erfolgt die Berechnung nach dem in Abschnitt 7.1 beschriebenen Verfahren.

`SetzeBildschirm` bestimmt die Abbildung vom NPC in das Bildschirmkoordinatensystem. Die Eingangswerte `x,y,Breite` und `Höhe` geben die Koordinaten und Auflösung des Bildschirmfensters in Pixeln an. Der Bereich $x_{NPC} = [0, w]$ des normierten Projektionskoordinatensystems wird dann auf (`x-0.5, x+Breite-0.5`) und entsprechend y_{NPC} auf (`y+Höhe-0.5, y-0.5`) abgebildet. Im Gegensatz zum NPC, in dem $\begin{pmatrix} x=0 & y=0 \end{pmatrix}$ „unten links" ist, befindet sich auf dem Bildschirm das Pixel mit den Mittelpunktskoordinaten $\begin{pmatrix} 0 & 0 \end{pmatrix}$ oben links. Die Addition von -0.5 bei der Abbildung ist die Folge der endlichen Auflösung der Pixel. Das Pixel mit den Mittelpunktskoordinaten $\begin{pmatrix} 0 & 0 \end{pmatrix}$ belegt z.B. den Bereich $\begin{pmatrix} -0.5 \ldots 0.5 & -0.5 \ldots 0.5 \end{pmatrix}$. Würde man diese Tatsache ignorieren, so wäre nicht sichergestellt, daß die Pixel an den Rändern des Bildschirmfensters gesetzt würden. Diese Betrachtungen sind erforderlich, da beim Rendern auch der Nachkommaanteil der Koordinaten berücksichtigt wird.

Um zu vermeiden, daß durch Rechenungenauigkeiten nach der Transformation in Bildschirmkoordinaten nicht Werte < -0.5 auftreten, welche zur Folge hätten, daß Pixel außerhalb des Darstellungsfensters gesetzt würden, wird die Verschiebung um ein ϵ korrigiert.

`SetzeBildschirm` bildet den Bereich $z_{NPC} = [0, w]$ zur weiteren Verwendung durch den z-Buffer-Algorithmus auf den Bereich $z = [0, 32767]$ ab. Dies ermöglicht die Verwendung von 16-Bit Integer-Zahlen für den z-Buffer. Letztlich hat die Matrix `NPC_DC` die Form

$$NPC_DC = \begin{pmatrix} 1 & x-(0.5-\epsilon) & y+(\text{Höhe}-1)+(0.5-\epsilon) & 32767 \\ 0 & \text{Breite}-2\epsilon & 0 & 0 \\ 0 & 0 & -\text{Höhe}-2\epsilon & 0 \\ 0 & 0 & 0 & -32767 \end{pmatrix}$$

Die Hilfsfunktion `IstRueckflaeche` berechnet nach dem in Abschnitt 7.3 beschriebenen Verfahren, ob eine Fläche Rückfläche ist. Zu diesem Zweck muß der Normalenvektor `n` der Fläche und ein Punkt `p` auf der Fläche sowie die Matrix `NPC_WC` übergeben werden. Wie alle Matrizen der Hilfsroutinen wird `NPC_WC` als Variablenparameter übergeben. Der Grund ist nicht, daß die Matrix in der Routine verändert wird, sondern daß die Übergabe von variablen Parametern sehr viel schneller ist als die Übergabe als Wertparameter.

Die Prozedur `WCnachNPCUndClipCode` transformiert den Punkt `p3`, nachdem er mit der homogenen Komponente $w = 1$ erweitert wurde, mit Hilfe der Matrix `aktWC_NPC` in das NPC und bestimmt anschließend den Clipcode des Punktes. Das Ergebnis ist der Punkt `p4` sowie ein Byte, dessen untere sechs Bit angeben, wie der Punkt relativ zu den Flächen $x = w$ (Bit 0), $x = 0$ (Bit 1), $y = w$ (Bit 2), $y = 0$ (Bit 3), $z = w$ (Bit 4) und $z = 0$ (Bit 5) steht. Dabei deutet ein gesetztes Bit an, daß der Punkt für die entsprechende Begrenzungsebene außerhalb des Sichtbereiches liegt.

Die letzte Hilfsroutine `NPCnachDC` transformiert einen Punkt `pNPC` aus homogenen NPC-Koordinaten mit Hilfe von `aktNPC_DC` in Bildschirmkoordinaten. In der Routine wird die Division durch w vorgenommen.

9.3 Die Unit `PRIMREND`, die Rendering-Pipeline aus Programmsicht

So geradlinig, wie die in Kap. 7 beschriebene Rendering-Pipeline erscheint, ist sie in der Implementierung nicht. Der Grund hierfür liegt im wesentlichen darin, nicht zuviel berechnen zu wollen. Warum sollten Punkte von Flächen, die gar nicht zu sehen sind, beleuchtet werden? Warum sollte ein Punkt, der Eckpunkt von 20 Flächen ist, 20 Mal beleuchtet werden? Um diese Verschwendung an Rechenzeit zu vermeiden, wollen wir im folgenden die ursprüngliche Rendering-Pipeline schrittweise modifizieren. Sie hat die in Abb. 9.1 dargestellte Form und wird der Reihe nach für alle Flächen der Szene durchlaufen.

Als ersten Optimierungsschritt führt man möglichst am Anfang der Pipeline das Backface-Culling, also das Entfernen von Rückflächen, ein. Dadurch entfällt i.a. für gut die Hälfte aller Flächen das Durchlaufen der restlichen Pipeline.

Als nächster Schritt in Richtung Optimierung trennen wir Punkt- und Flächenberechnung. Mit diesem Schritt ist es dann möglich, das weiter oben erwähnte n-fache Berechnen für einen Eckpunkt zu vermeiden.

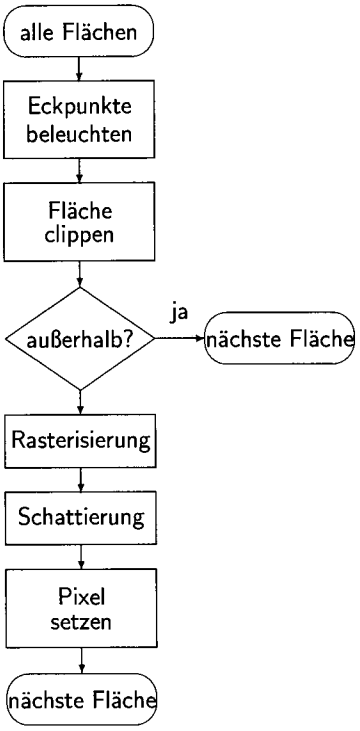


Abb. 9.1 Die Rendering-Pipeline in der ursprünglichen Form.

Da die Punkte nach dem Einlesen ohnehin getrennt von den Flächen gespeichert sind (die Flächen greifen indirekt über Indizes auf die Punkte zu), ist es einfach, eine zweite Pipeline für Punkte einzuführen. Wir haben diese Pipeline „Punkt-Pipeline" getauft.

Genauso wie die Rückflächen ganz am Anfang der Pipeline ausgeschlossen werden, schließt man außerhalb des Sichtbereichs liegende Punkte ebenfalls zu Beginn der Punkt-Pipeline von weiteren Berechnungen aus. Nun kann es passieren, daß solche Punkte trotzdem bei der Darstellung von teilweise sichtbaren Flächen benötigt werden. In diesem Fall muß dann der entsprechende Punkt eben noch nachbehandelt werden. Um Punkte wie beschrieben ausschließen zu können, ist der erste Schritt der Punkt-Pipeline das Berechnen der Clipcodes der Punkte. Diese Clipcodes können nach dem Berechnen der Punkt-Pipeline auch von der Flächen-Pipeline benutzt werden, um zu entscheiden, ob eine Fläche überhaupt abzuschneiden ist.

Für Punkte, die im Inneren des Sichtbereichs liegen, erfolgt anschließend die Beleuchtungsberechnung (siehe Kap. 10). Flächen, an denen ein Punkt als Eckpunkt beteiligt ist, haben i.a. verschiedene Reflektivitäten. Aus diesem Grund verzichtet man bei der Beleuchtung auf die Multiplikation der Flächenreflektivität mit der einfallenden diffusen Intensität und vollzieht diese Multiplikation erst, wenn die entsprechende Fläche dargestellt werden soll.

Ein Beispiel: Ein rot-weißes Schachbrett ist als Menge von Vierecken abgespeichert. Die Punktliste beinhaltet alle Eckpunkte der verschiedenen Vierecke nur einmal. Bei der Beleuchtungsberechnung ergibt sich z.B. für alle Eckpunkte die einfallende diffuse „Farbe" $\begin{pmatrix} 0.9 & 0.8 & 0.7 \end{pmatrix}$. Bei der Darstellung der roten Flächen wird nun diese Farbe mit $\begin{pmatrix} 1 & 0 & 0 \end{pmatrix}$ multipliziert, bei der Darstellung der weißen mit $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$, so daß letztlich Flächen mit den Farbwerten $\begin{pmatrix} 0.9 & 0 & 0 \end{pmatrix}$ und $\begin{pmatrix} 0.9 & 0.8 & 0.7 \end{pmatrix}$ dargestellt werden.

Diese Vorbeleuchtung macht natürlich nur Sinn, wenn ein Punkt für alle an ihn grenzenden Flächen die gleichen Eigenschaften hat. Dieses ist bei „scharfen" Punkten nicht der Fall. Scharfe Punkte sind Punkte, die zwar für verschiedene Flächen benutzt werden, da sich diese Flächen eine Ecke teilen, die aber je nach Fläche verschiedene Normalenvektoren haben.

Für diese scharfen Punkte macht die Beleuchtung im voraus keinen Sinn. Für sie gilt das gleiche wie für außerhalb liegende Punkte, die trotzdem benötigt werden. Die Beleuchtung erfolgt flächenabhängig nach Bedarf.

Mit diesen Beschreibungen ergeben sich zwei Pipelines – die Punkt- und die Flächen-Pipeline – die in Abb. 9.2 dargestellt sind. In der Punkt-Pipeline werden alle Punkte beleuchtet, die nicht außerhalb oder „scharf" sind. Genauso, wie etwa die Hälfte aller Flächen deswegen nicht dargestellt werden, weil sie Rückflächen sind, ist es auch nicht nötig, für die Eckpunkte dieser Flächen den Clipcode zu bestimmen und sie zu beleuchten, falls sie nicht zusätzlich Eckpunkte von Vorderflächen sind. Auf diese Art ist es möglich, etwa die Hälfte der Punkte von der Clipcodeberechnung und Beleuchtung auszuschließen.

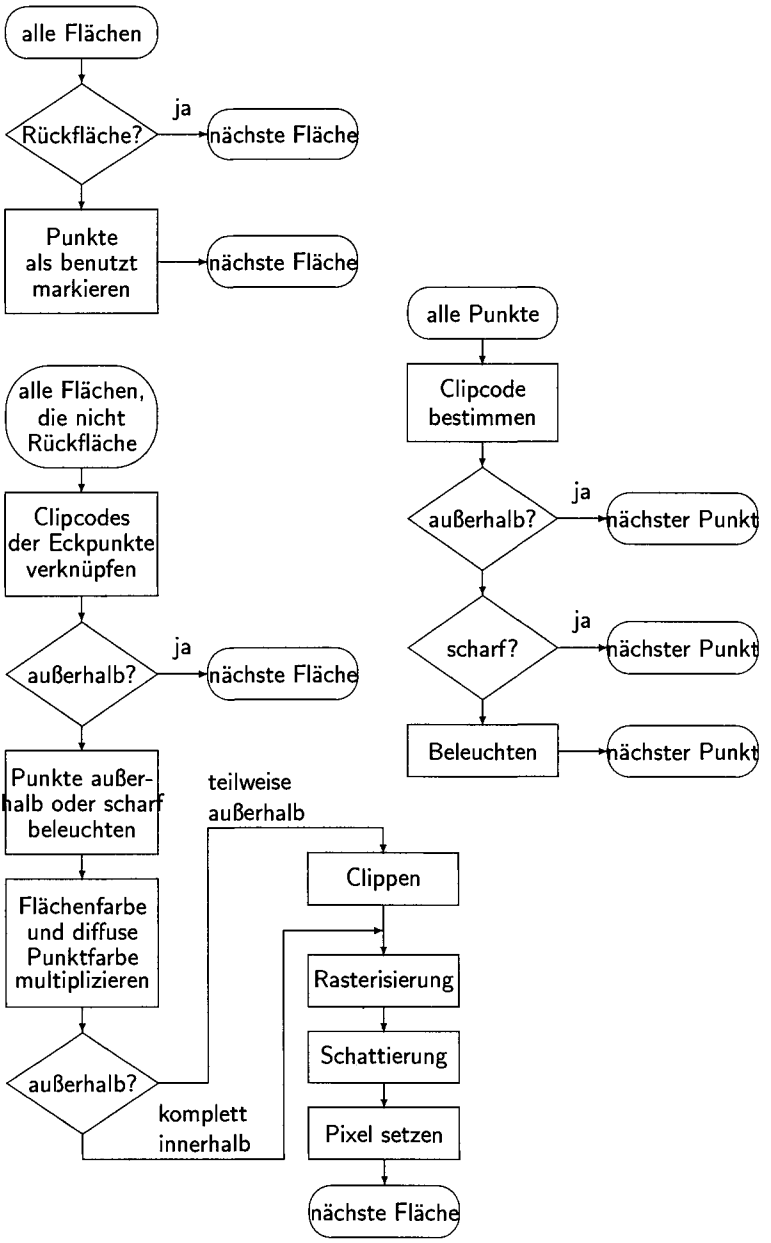


Abb. 9.2 Die in Punkt- und Flächen-Pipeline unterteilte Rendering-Pipeline. Beim Rendering wird die Punkt-Pipeline zwischen dem ersten und dem zweiten Teil der Flächen-Pipeline durchlaufen.

Um festzustellen, ob ein Punkt ausschließlich Eckpunkt von Rückflächen ist, muß das Backface-Culling für die Flächen *vor* der Punkt-Pipeline durchlaufen werden. Alle Punkte von Vorderflächen werden in diesem Durchlauf mit `PunktTYP.benutzt` markiert. Nur diese benutzten Punkte werden dann in der Punkt-Pipeline behandelt. Ob Flächen Rückflächen sind, wird in `GlobaleFlaecheTYP.IstRueckflaeche` gespeichert.

Hiermit läßt sich die Rendering-Pipeline als Algorithmus zusammenfassen.

Algorithmus 9.1 (Optimierte Rendering-Pipeline)

FÜR alle Flächen:
  Berechne, ob Fläche Rückfläche.
  Speichere das Ergebnis in der Fläche.
  WENN Fläche Vorderfläche, DANN:
    Markiere Eckpunkte als benutzt.
  ENDE WENN.
ENDE FÜR alle Flächen.
FÜR alle Punkte, die als benutzt markiert sind:
  Bestimme Clipcode des Punktes und speichere ihn.
  WENN ClipCode $\neq$ außerhalb, DANN:
    WENN Punkt nicht scharf ist, DANN:
      Beleuchte den Punkt.
    ENDE WENN Punkt nicht scharf ist.
  ENDE WENN ClipCode $\neq$ außerhalb.
ENDE FÜR alle Punkte.
FÜR alle Flächen:
  WENN die Fläche nicht Rückfläche, DANN:
    Verknüpfe die Clipcodes aller Eckpunkte der Fläche.
    WENN die Fläche nicht komplett außerhalb, DANN:
      FÜR alle Punkte der Fläche, die außerhalb..
      .. oder scharf sind:
        Beleuchte den Punkt.
      ENDE FÜR alle Punkte.
      Multipliziere die Reflektivität der Fläche mit der..
      .. diffusen Farbe des Punktes.
      Stelle die Fläche dar.
    ENDE WENN die Fläche nicht komplett außerhalb.
  ENDE WENN die Fläche nicht Rückfläche.
ENDE FÜR alle Flächen.

In der Unit `PRIMREND` ist nur eine einzige Prozedur, nämlich `RendereSzene` definiert. Sie implementiert den gerade vorgestellten Algorithmus.

Da an den verschiedenen Stellen des Algorithmus diverse Parameter benötigt werden, ist der Prozedurkopf von `RendereSzene` entsprechend umfangreich:

```
PROCEDURE RendereSzene(Lichtquellen: LichtquellenPtrTYP;
                       Betrachtung: BetrachterTYP;
                       WC_NPC, NPC_WC, NPC_DC, WC_DC: Mat4TYP;
                       Hintergrund: HintergrundTYP;
                       AnzahlPunkte, AnzahlFlaechen: LONGINT;
                       FarbeVorberechnet: BOOLEAN;
                       DiffVorZuDiff, SpekGewichtung: SINGLE;
                       VorBerechnetesAmbiente: FarbVektorTYP);
```

Die `Lichtquellen` und die `Betrachtung` werden zum Beleuchten benötigt. Die Transformationsmatrizen `WC_NPC`, `NPC_WC`, `NPC_DC`, `WC_DC` dienen der Transformation der Punkte beim Clippen sowie der Transformation in Bildschirmkoordinaten. Mit Hilfe von `NPC_WC` wird festgestellt, ob eine Fläche Rückfläche ist. Der `Hintergrund` wird vor der Darstellung der Flächen gezeichnet. Er wird direkt aus der Beschreibung der Szene übernommen.

`AnzahlPunkte` und `AnzahlFlaechen` geben an, wieviele Punkte und Flächen die Szene enthält. Auf diese wird mit Hilfe spezieller Routinen (siehe `PRIMMEM`) zugegriffen. Die restlichen Parameter dienen zum Mischen von Radiosity-Vorbeleuchtung und „normaler" Beleuchtung. Diese Parameter werden im Zusammenhang mit `StelleFlaecheDar` der Unit `PRIMSHD` (siehe Abschnitt 15.4.2) genauer erläutert.

Die Zeilen 20...27, also der Teil des Algorithmus, der für Flächen, die tatsächlich dargestellt werden, durchlaufen wird, erledigt `RendereSzene` durch den Aufruf der Prozedur `StelleFlaecheDar`.

Zusätzlich zum obigen Algorithmus ist in `RendereSzene` noch eine Besonderheit implementiert, die im Zusammenhang mit dem Radiosity-Verfahren nötig ist: Durch die Unterteilung der Flächen für die Energieberechnung (Abschnitt 18.4) kann es passieren, daß an der Kante einer Fläche zwei Nachbarflächen existieren. Diese Kante ist also in der Mitte geteilt – ein sogenanntes T-Stück. Bei der Schattierung ist dann die Farbverteilung entlang dieser Kante nicht mehr stetig (Abb. 9.3). Bei der Darstellung solcher Unstetigkeiten würden im Bild Kanten erscheinen, wo keine sind. Um diesen Effekt zu vermeiden, werden Flächen in Unterflächen zerteilt, wenn ihre Kanten unterteilt sind.

Da nach dem Generieren der Punkt- und Flächenlisten alle darzustellenden Flächen entweder Drei- oder Vierecke sind, können nur die in Abb. 9.4 dargestellten Unterteilungsfälle auftreten. Je nach Anzahl der unterteilten Kante wird eine Fläche also in 1–4 Unterflächen zerlegt. Wie die Nachbarstruktur für Flächen aussieht und zu welchem Zweck sie gebraucht wird, kann genauer im Abschnitt 18.4.2 nachgelesen werden.

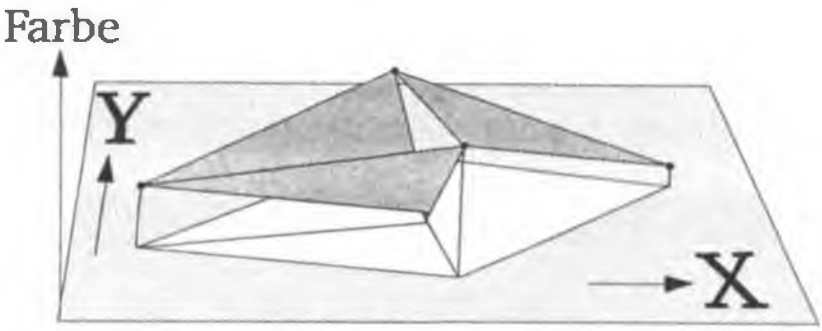


Abb. 9.3 Unstetigkeit in der Farbe bei einem T-Stück

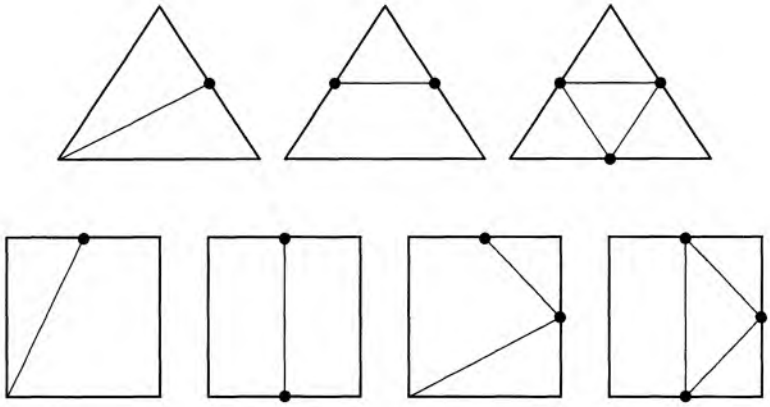


Abb. 9.4 Mögliches Auftreten von T-Stücken bei Drei- und Vierecken und daraus resultierende Unterteilungen

10. Modelle für die Beleuchtungsrechnung

Eine für die realistische Darstellung einer Szene extrem wichtiger Punkt ist die Simulation der Beleuchtungsverhältnisse. Gemeinsam mit der Verdeckung macht sie den Großteil dessen aus, was als realistisch an einem Bild empfunden wird. Um die Beleuchtung simulieren zu können, sind verschiedene Modelle zu entwickeln:

- ein *Farbmodell* hilft, mit Farben rechnen zu können,
- *Beleuchtungsmodelle* sind Vorschriften, wie Intensitäten (oder einfacher gesagt: Farben) an bestimmten Stellen einer Szene berechnet werden können und
- *Lichtquellenmodelle* helfen, bestimmte Lichtverhältnisse erst herzustellen, indem Lichtquellen mit bestimmten Eigenschaften in einer Szene plaziert werden.

Fast alle diese Modelle orientieren sich an physikalischen Gesetzmäßigkeiten, sind aber teilweise auch empirisch begründet. Wollen wir uns nicht so sehr auf eine Simulation der physikalischen Eigenschaften festlegen, genügen im allgemeinen sehr einfache Modelle, um „schöne" Bilder erzeugen zu können.

Lokale Beleuchtungsmodelle sind z.B. solche „einfachen Modelle". Sie berücksichtigen keine Intensitätsübergänge zwischen Objekten und sind daher besonders schnell berechenbar. Mit globalen Beleuchtungsmodellen sind aufwendiger realistischere Bilder zu erzielen. Sie sind Grundlage der Strahlverfolgungs- und Strahlungsverfahren.

10.1 Farben

Licht ist eine elektromagnetische Welle. Die vom Auge wahrgenommene Farbe hängt von der Wellenlänge des Lichtes ab. Im allgemeinen treten dabei Farben nicht als „reine" Farben genau einer Wellenlänge auf (monochrom), sondern sie sind Mischungen mehrerer dieser Farbanteile. Theoretisch kann gezeigt werden, daß sich alle Farben als Linearkombination dreier Grundfarbspektren zusammensetzen lassen. Das bedeutet, daß eine Farbe als ein Vektor

$$C = \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

darstellbar ist. Die Komponenten wurden 1931 von einer internationalen Kommission festgelegt.

Sie sind ebenfalls keine monochromen Farben, sondern Farbspektren. Eine Vereinfachung dieses Modells besteht nun darin, Farben einer Wellenlänge für die drei Primärkomponenten anzunehmen. Das am häufigsten dafür gewählte Modell ist das *RGB-Modell*, in dem die drei Primärkomponenten die Farben Rot, Grün und Blau sind. Die Folgen dieser Vereinfachung sind:

1. Es geht Farbinformation im Bild verloren.
2. Die Werte sind aufgrund der Gerätetechnologie gewählt und sind daher nicht an den Bedürfnissen der Beleuchtungsberechnung orientiert.
3. Die Darstellung dieser Farben ist geräteabhängig. Ergebnisse von Berechnungen, die prinzipiell die gleiche Farbe darstellen sollen, werden auf unterschiedlichen Geräten unterschiedlich wahrgenommen.

In der grafischen Datenverarbeitung werden noch weitere dreiwertige Modelle verwendet [Cla90].

CMY: Statt des additiven Farbmodells definiert CMY mit den Primärfarben cyan (ein Blauton), magenta (Tiefrot) und yellow (Gelb) ein subtraktives Farbmodell. Dieses Farbmodell eigenet sich vor allem für die Farbbeschreibung bei farbauftragenden Geräten oder in der Fototechnik. Sowohl RGB als auch CMY werden durch einen Einheitswürfel geometrisch repräsentiert.

HSV: Basiert auf den drei Primärkomponenten Farbton (hue), Sättigung (saturation) und Intensität (value). Dieses Farbmodell entspricht in etwa der Art, wie Maler Farben mischen und kann in einem Kegel geometrisch dargestellt werden.

HLS: Statt der Intensität wird bei diesem Farbmodell die Helligkeit (lightness) zusammen mit dem Farbton und der Sättigung verwendet. Es wird in einem Doppelkegel geometrisch dargestellt.

Eine ausführliche Beschreibung dieser Modelle kann z.B. bei Roy Hall nachgelesen werden [Hal88]. Für die Zwecke dieses Buches soll die Verwendung der drei Primärfarben Rot, Grün und Blau genügen. Unter einer Farbe oder Intensität wird also ein Vektor

$$I = C = \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

verstanden.

10.2 Lichtquellen

Eine Lichtquelle zeichnet aus, daß sie eine gewisse Intensität abstrahlt, die dann Objekte in einer Szene beleuchtet. Im folgenden stellen wir die gängigsten Lichtquellenmodelle vor. Sie stammen größtenteils aus PHIGS PLUS (Programmer's Hierarchical Interface to Graphic Systems – Plus Lumiere Und Shading), einem internationalen Standardisierungsvorschlag [vD88].

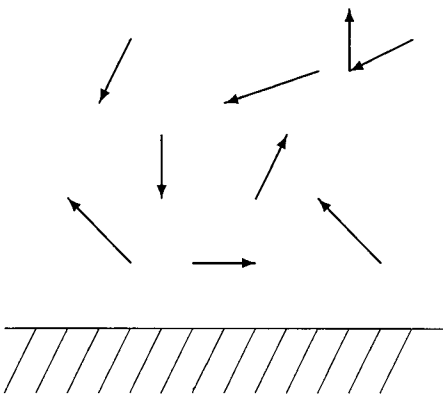


Abb. 10.1 Ambiente Lichtquelle

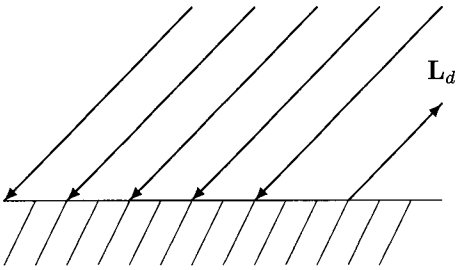


Abb. 10.2 Richtungslichtquelle

In einer *ambienten Lichtquelle* werden alle indirekten Beleuchtungen einer Szene zusammengefaßt. Selbst wenn Lichtquellen verdeckt sind oder nur indirekte Beleuchtungen in einer Szene vorhanden sind, erscheinen Objekte ja in einer bestimmten Grundhelligkeit. Werden diese indirekten Beleuchtungen im Algorithmus nicht berücksichtigt, so wird ambientes Licht mit einer bestimmten Intensität angenommen. Im Strahlverfolgungsverfahren und im Radiosity-Verfahren kann es nützlich sein, eine solche Lichtquelle zu verwenden, es ist im Prinzip aber nicht nötig.

Ambiente Lichtquellen spenden Licht einer bestimmten Intensität I_a zu allen Punkten einer Szene (Abb. 10.1). Sind mehrere dieser Lichtquellen vorhanden, können sie zu einer zusammengefaßt werden. In Abb. III (im farbigen Bildteil) ist die Wirkung verschiedener Lichtquellen auf einfache Objekte gezeigt.

Ein zweites Lichtquellenmodell berücksichtigt nicht nur die Intensität der Lichtquelle, sondern auch die Richtung, aus der diese Lichtquelle scheint. Sie ist damit eine Lichtquelle „im Unendlichen". Als Beispiel für diese Lichtquelle kann annähernd Sonnenlicht auf der Erde angesehen werden. Unendlich weit entfernte Lichtquellen werden oft hinter dem Betrachterstandpunkt plaziert, um die Darstellung von Schatten im Bild zu vermeiden.

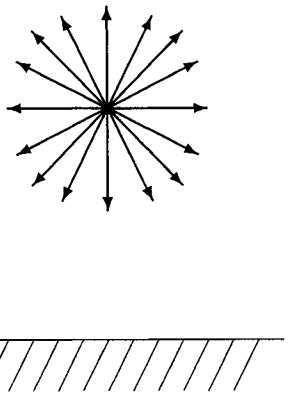


Abb. 10.3 Punktlichtquelle

Richtungslichtquellen leuchten Licht einer bestimmten Intensität I_d in eine bestimmte Richtung $-L_d$ zu allen Punkten einer Szene (Abb. 10.2 und III im farbigen Bildteil). Sind mehrere dieser Lichtquellen vorhanden, so können sie nicht zusammengefaßt werden, sondern die Beiträge zur Beleuchtung der Szene überlagern sich. Eine Überlagerung heißt konkret: die Addition einzelner Beleuchtungsanteile.

Die nächste Kategorie von Lichtquellen wird als *Punktlichtquelle* bezeichnet. Punktlichtquellen sind im Raum plaziert und infinitesimal klein. Sie strahlen eine bestimmte Intensität in alle Richtungen des Raumes ab. Sie ähneln einer Glühbirne, die im Lichtkegel nicht beschränkt ist (Abb. 10.3).

Eine Punktlichtquelle wird also durch den Punkt P_p, an dem sie lokalisiert ist, und die Intensität I_p, die sie abstrahlt, charakterisiert. Eine Beleuchtungsrichtung muß für jeden Punkt einer Szene neu ausgerechnet werden. Der Effekt von Punktlichtquellen ist eine begrenzte Beleuchtung in der „Nähe" der Position der Lichtquelle (Abb. III im farbigen Bildteil). Die Verwendung von Punktlichtquellen wird im Strahlverfolgungsverfahren auf „Kugellichtquellen" mit sehr kleinem Durchmesser erweitert. Bei Beibehaltung der Punktförmigkeit würden wohl die wenigsten Strahlen eine Lichtquelle jemals treffen.

Die *Spotlichtquelle* ist eine Variation der Punktlichtquelle. Sie wird mit zusätzlichen Attributen versehen, die das Verhalten realer Lichtquellen besser annähern. Das erste Charakteristikum ist, daß ein Strahler nur in einem begrenzten Kegel Licht emittiert. Dieser Kegel wird durch den Ausbreitungswinkel α festgelegt (Abb. 10.4).

Er ist um die Hauptausstrahlungsrichtung $-L_s$ der Spotlichtquelle herum lokalisiert. Ein weiteres Attribut ist der sogenannte Konzentrationsexponent c_s, der die Intensität I_s des Strahlers variiert. Der Effekt ist, daß in der Hauptausstrahlungsrichtung die größte Intensität, in Richtung der Ränder des Ausstrahlungskegels geringere Intensität abgestrahlt wird. Der Strahlungskegel ist auf beleuchteten Objekten deutlich wahrnehmbar (siehe Abb. III).

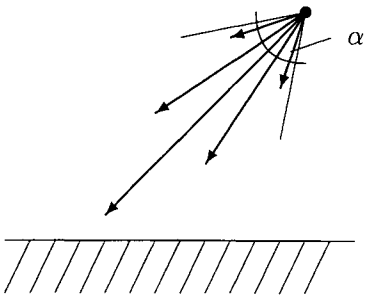


Abb. 10.4 Spotlichtquelle

Gemäß der Annäherung von physikalischen Objekten durch Flächenstücke kann natürlich auch eine beliebig geformte Lichtquelle durch kleine Flächenlichtquellen angenähert werden. Ändert sich die abgestrahlte Intensität innerhalb einer solchen Flächenlichtquelle nicht, wird sie als homogen bezeichnet.

Für eine *homogene Flächenlichtquelle* sind die Intensität I_h, die Position des Mittelpunktes P_h, die Fläche der Lichtquelle A_h und die Flächennormale und damit die Ausstrahlungsrichtung $\mathbf{N}_h$ charakterisierende Merkmale. Flächenlichtquellen werden üblicherweise nur in Strahlungs- (Radiosity) oder Strahlverfolgungsverfahren eingesetzt, nicht aber im Rendering mit Polygonen.

10.3 Abschwächung der Intensität durch Entfernung

Die Intensität des Lichtes schwächt sich bei zunehmender Entfernung von der Lichtquelle ab. Das ist darauf zurückzuführen, daß bei gleichbleibender Strahlungsleistung $\phi[Watt]$ einer Lichtquelle die Intensität als das Differential

$$I = \frac{d\phi}{d\omega}$$

beschrieben wird. $d\omega$ bezeichnet den Raumwinkel um die Position der Lichtquelle. Ist eine beleuchtete Fläche in der Entfernung r von der Lichtquelle plaziert, so kann der Raumwinkel durch

$$\omega = \frac{A \cdot \cos(\theta)}{r^2}$$

angenähert werden. A ist der Flächeninhalt des bestrahlten Objektes und θ der Winkel des einfallenden Lichtes. Die Intensitätsabschwächung zwischen Lichtquelle und Fläche ist also umgekehrt proportional zu r^2 (Abb. 10.5).

$$I_{Fläche} \sim \frac{1}{r^2} \cdot I_{Lichtquelle}$$

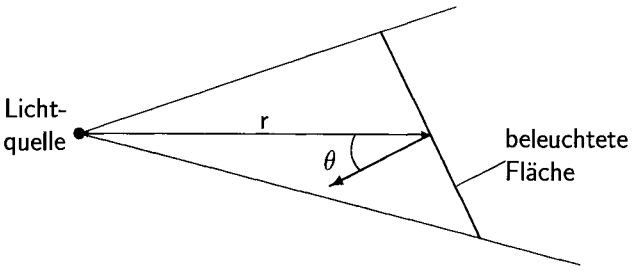


Abb. 10.5 Geometrie der Abschwächung durch die Entfernung

Die Abschwächung durch Entfernung wird oft nicht durch Gewichtung mit dem Faktor $1/r^2$ berechnet. Alternativ werden $1/r$ oder $1/(a+br)$ verwendet (z.B. in PHIGS PLUS) [vD88]. Die Parameter a und b werden in Kap. 11.2 näher erläutert.

10.4 Abschwächung mit Nebeleffekt

Soll Nebel als Effekt simuliert werden, so kann dies ebenfalls mittels einer Abschwächung der Intensität der Lichtquelle in Abhängigkeit von der Entfernung erfolgen. Im Gegensatz zur oben angesprochenen Abschwächung addiert sich aber zur Intensität, die die Objekte direkt erreicht, eine Intensität, die vom Nebel herrührt. Dieser wird selbst beleuchtet und die in ihm enthaltenen (Wasser)-Teilchen ihrerseits beleuchten die Umgebung. Man spricht hier von sogenanntem scattering.

In der Praxis wird dieser Effekt simuliert, indem man die durch direkte Beleuchtung entstehende Objektfarbe mit einer „Nebelfarbe" mischt. Dabei hängt das Mischungsverhältnis vom Abstand des Objektes zum Auge ab.

Um die Dichte des Nebels beeinflussen zu können – z.B. für Nebelbänke – erfolgt die Mischung nur innerhalb eines Tiefenintervalls $[l_{min}, l_{max}]$. Außerhalb dieses Bereiches scheinen Objekte konstant „vernebelt".

Bezeichnet I_O die Objektfarbe nach der normalen Beleuchtung und I_N die Nebelfarbe, dann ergibt sich die darzustellende Intensität zu:

$$
\begin{aligned}
I &= I_O \cdot k_{dc} + I_N \cdot (1 - k_{dc}) \qquad \text{mit} \\
k_{dc} &= \begin{cases} k_{min}, & l < l_{min} \\ k_{max}, & l > l_{max} \\ k_{min} + \frac{(l - l_{min})(k_{max} - k_{min})}{l_{max} - l_{min}}, & \text{sonst} \end{cases} \qquad \text{und} \\
0 &\leq k_{min} < k_{max} \leq 1.
\end{aligned}
$$

wobei l die Entfernung des Objektes zum Betrachtungspunkt und k der Gewichtungsfaktor der Nebelfarbe ist.

10.5 Abschwächung der Intensität in Medien

Durch Lichtquellen wird Licht einer bestimmten Intensität an der Position der Lichtquelle erzeugt. Die auf dem Bildschirm wahrnehmbare Farbe wird aber im allgemeinen nicht direkt von dieser Lichtquelle hervorgerufen. Lichtquellen bestrahlen Objekte, die Licht dann in einer für sie charakteristischen Weise weiter verbreiten. Durch Objekte wird also die Intensität geändert. Ein weißes Objekt gibt beispielsweise jede Intensität wie empfangen weiter. Wird dagegen ein grünes Objekt mit rotem Licht bestrahlt, so erscheint es schwarz.

Eine weitere Veränderung der Intensität tritt infolge von Medien auf, die das Licht dämpfen. Je weiter der Weg des Lichtes durch das dämpfende Medium ist, desto geringer wird die Intensität des Lichtes. Physikalisch wird vom Phänomen der „Absorption" des Lichtes gesprochen. Es wird dabei z.B. in Wärme umgewandelt. Gemäß Absorptionsgesetz gilt, daß die Abschwächung des Lichtes in einem Medium von der eingedrungenen Intensität, dem zurückgelegten Weg und dem Absorptionskoeffizienten des Mediums abhängt.

$$I = I_0 \cdot e^{-al}$$

I ist die beobachtete Farbe oder Intensität, I_0 die in das Medium eingedrungene Intensität, l der im Medium zurückgelegte Weg und a der Absorptionskoeffizient. Dieser ist immer größer als Null. Durchsichtige Körper haben niedrige Absorptionskoeffizienten, nicht durchscheinende relativ hohe. Das Modell wird nicht nur innerhalb von (festen) Körpern angewendet, sondern beispielsweise auch innerhalb der die Körper umgebenden Luft.

Für die praktische Umsetzung ist diese Berechnung auf allen „Lichtwegen" allerdings zeitaufwendig, so daß oft mit vereinfachten Methoden, wie im Abschnitt über Abschwächung durch Entfernung beschrieben, gearbeitet wird. Zwischen Lichtquellen und Objekten wird das Licht mit dem Kehrwert des Abstandes, eventuell auch mit dem Quadrat des Abstandes, in der Intensität geschwächt.

10.6 Oberflächen

Viel gravierendere Veränderungen des Lichtes und seiner Intensität findet bei der Interaktion von Licht an Grenzflächen zwischen zwei Medien statt. Diese Grenzflächen sind beliebig dünn. An ihnen können zwei Effekte eintreten: das Licht wird reflektiert oder es wird transmittiert. Bei einer *Reflexion* übertritt das Licht die Grenzfläche nicht, verbleibt also im Medium, aus dem es auf die Grenzfläche trifft. Die *Transmission* bewirkt ein Überschreiten der Grenzfläche und damit ein Eindringen des Lichtes in ein neues Medium.

Im allgemeinen wird das auf eine Grenzfläche treffende Licht in Bruchteilen reflektiert und der Rest entsprechend transmittiert. Diese Aufteilung wird mit einem Transmissions- bzw. Reflexionsgrad modelliert. Die Richtung, in der das Licht reflektiert bzw. transmittiert wird, ergibt sich aus dem Reflexions-

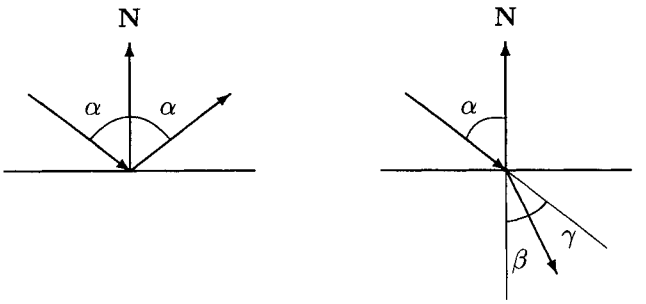


Abb. 10.6 Geometrie des Reflexions- und Brechungsgesetzes

und Brechungsgesetz (Abb. 10.6). Das erste besagt, daß der Einfallswinkel des Lichtes gleich dem Ausfallswinkel des Lichtes ist. Im zweiten wird eine Beziehung zwischen dem Einfalls- und dem Brechungswinkel hergestellt. Mit diesen Vorschriften wird eine neue Richtung des Lichtes festgelegt, nicht aber, wie sich die Intensität des Lichtes an der Grenzfläche verändert. Dies ist in den folgenden Formeln und Modellen beschrieben.

Da für das Licht das Superpositionsgesetz gültig ist, können sämtliche der im folgenden dargestellten Effekte aus Linearkombinationen von Einzeleffekten berechnet werden. Beobachtet man beispielsweise an einer Grenzfläche ambiente, diffuse und spiegelnde Teilintensitäten $I_{amb}, I_{diff}, I_{spec}$, so ergibt sich für die Gesamtintensität

$$I = I_{amb} + I_{diff} + I_{spec}.$$

Die Art und Weise der Zusammenstellung der verschiedenen Modelle ist Inhalt des verwendeten Beleuchtungsalgorithmus.

Zuerst betrachten wir die Reflexion ambienten Lichtes.

Ungerichtetes Licht trifft aus allen Richtungen gleichmäßig auf eine Fläche auf und wird dementsprechend auch gleichmäßig wieder in alle Richtungen reflektiert. Der Reflexionsgrad k_a bezeichnet den Anteil des eintreffenden Lichtes, der wieder reflektiert wird. Er ist wellenlängenabhängig und liegt zwischen Null und Eins. Somit ist die Intensität der an einem Objekt in alle Richtungen auftretenden Reflexion ambienten Lichtes:

$$I = k_a \cdot I_a = \begin{pmatrix} r_a I_{ar} \\ g_a I_{ag} \\ b_a I_{ab} \end{pmatrix}$$

Die Multiplikation zwischen den Farbvektoren k_a und I_a ist dabei als komponentenweise Multiplikation zu interpretieren. Dies entspricht dem physikalischen Effekt, daß beispielsweise ein grün bestrahlter roter Körper schwarz erscheint, also kein Licht reflektiert.

Mit Ausnahme des ambienten Lichtes ist für das Licht eine bestimmte Richtung, in die es sich ausbreitet, gegeben. An einer Oberfläche kann es reflektiert oder transmittiert werden. Bei der Reflexion wie bei der Transmission werden

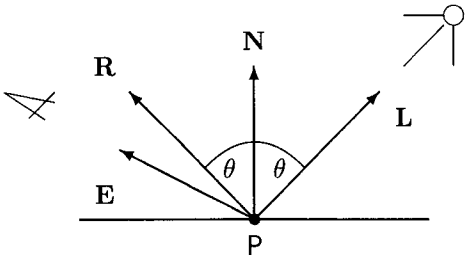


Abb. 10.7 Geometrie der Beleuchtungsmodelle

Tabelle 10.1 Beschreibung der Parameter der Beleuchtungsmodelle

Bezeichnung	Bedeutung
P	ein auf einer Grenzfläche betrachteter Punkt
E	der normierte Vektor vom Punkt P in Richtung des Beobachtungspunktes
N	der Normalenvektor der Grenzfläche im Punkt P
L	der normierte Vektor vom Punkt P in Richtung der Lichtquelle
R	der reflektierte Lichtquellenvektor

diffuse und spiegelnde Verbreitung unterschieden. Eine diffus reflektierende Grenzfläche streut das Licht in alle Richtungen und erscheint matt. Spiegelnde Flächen hingegen haben eine ausgeprägte Reflexion in die ideal reflektierte Richtung und erscheinen daher glänzend.

Für die im folgenden vorgestellten Modelle zur Berechnung der Intensität werden bestimmte Vektoren und Punkte benötigt, die in Abb. 10.7 dargestellt und in Tabelle 10.1 beschrieben sind.

Mit Hilfe einfacher Modelle lassen sich die Intensitäten, die vom Blickpunkt aus erkannt werden, berechnen. Eines der ältesten dieser Modelle ist das Modell von Gouraud [Gou71]. Es basiert auf dem Lambertschen Kosinusgesetz und besagt das folgende: Die beobachtete Intensität ist proportional zum Kosinus des Winkels zwischen der Normale und dem Lichteinfallsvektor. Ist die Fläche von der Lichtquelle abgewendet, so wird sie nicht beleuchtet.

Bemerkenswert ist, daß der Intensitätseindruck vom Blickpunkt unabhängig ist. Das ist auf die gleichmäßige Streuung des reflektierten Lichtes in alle Richtungen bei einer diffusen Reflexion zurückzuführen. Der Reflexionskoeffizient gibt in diesem Modell die Farbe an, in der ein Objekt unter weißem Licht erscheint. In Formeln ausgedrückt lautet das Gouraud-Modell:

$$I = \begin{cases} 0, & \cos(\theta) \leq 0 \\ k_d \cdot \cos(\theta) \cdot I_L, & \cos(\theta) > 0 \end{cases}$$

θ ist der Winkel zwischen der Normale und dem Lichtquellenvektor. Sind beide Vektoren normiert, d.h. sie haben die Länge 1, so kann der Kosinus des Winkels auch durch das Skalarprodukt zwischen diesen beiden Vektoren ersetzt werden.

$$I = \begin{cases} 0, & (\mathbf{N} \cdot \mathbf{L}) \leq 0 \\ k_d \cdot (\mathbf{N} \cdot \mathbf{L}) \cdot I_L, & (\mathbf{N} \cdot \mathbf{L}) > 0 \end{cases}$$

Diese Berechnung ist oft einfacher zu implementieren, hat aber den Nachteil, daß die Vektoren normiert werden müssen, bevor eine Farbe berechnet wird.

Bei einem Beleuchtungsmodell für spiegelnde Reflexion kann auf das Reflexionsgesetz zurückgegriffen werden (Abb. 10.6). Die Intensität, die eine Lichtquelle zu einer Oberfläche strahlt, wird um einen bestimmten Faktor gemindert und in Richtung des reflektierten Lichtquellenvektors $\mathbf{R}$ wieder abgestrahlt. Das bedeutet eigentlich, daß nur dann eine Intensität wahrgenommen werden kann, wenn der Blickpunkt auch genau in Richtung des reflektierten Lichtquellenvektors liegt.

Eine sinnvolle Erweiterung durch die Messung der Abweichung zwischen dem reflektierten Lichtquellenvektor und dem Blickvektor wurde 1975 durch Phong eingeführt [Pho75]. Dieses empirische Modell ergänzt die Parameter einer Fläche durch einen Helligkeitsexponenten ($m > 0$), der angibt, wie matt oder glänzend eine Oberfläche erscheinen soll. Die Intensität wird dann mit der folgende Beziehung berechnet:

$$I = \begin{cases} 0, & (\mathbf{E} \cdot \mathbf{R}) \leq 0 \\ k_s \cdot (\mathbf{E} \cdot \mathbf{R})^m \cdot I_L, & (\mathbf{E} \cdot \mathbf{R}) > 0 \end{cases}.$$

Den normierten, reflektierten Lichtquellenvektor erhält man mit folgender Formel:

$$\mathbf{R} = 2\mathbf{N}' - \mathbf{L} \qquad \text{mit} \qquad \mathbf{N}' = \mathbf{N}(\mathbf{N} \cdot \mathbf{L}).$$

Entsprechend wird mit den transmittierten Intensitäten verfahren.

Es existiert noch eine Vielzahl von weiteren Beleuchtungsmodellen, die alle mehr oder weniger verfeinert sind (nachzulesen z.B. in [FvDFH90]). Die Grundidee ist aber in allen gleich: Einzelne Effekte der Reflexion oder Brechung an einer Grenzfläche werden in einem Modell mit mehr oder weniger gut interpretierbaren Parametern abgebildet.

11. Beispiele zur Beleuchtungsberechnung

In diesem Kapitel wird gezeigt, welches Farbmodell, welche Lichtquellen- und welche Reflexionsmodelle in PRIM realisiert wurden.

11.1 Farbmodelle

Im beiliegenden Programm wurde das RGB-Modell implementiert. Soll die Farbe eines Objektes definiert werden, so geschieht das mit dem Schlüsselwort OD. OD bezeichnet die Farbe, die ein Objekt bei vollständig diffuser Reflexion erhält und damit das, was umgangssprachlich die Farbe eines Objektes ausmacht.

Bei polygonalen Teilobjekten kann jeder einzelnen Fläche übrigens mit der Angabe R eine Farbe gegeben werden (siehe Anhang B).

Greifen wir das einfache Beispel mit dem Molekülring aus Kap. 8 nochmals auf. Damit die einzelnen Atome deutlicher voneinander unterscheidbar sind, soll jedes eine andere Farbe erhalten.

```
GED_VERSION 1;

OBJEKT Atom;
  VERSCHIEBE 2,0,0;
  KUGEL;
ENDOBJEKT;

STRAHLER 1,1,1 0,0,-1 0,0,3 1,0 5 70;

SZENE Beispiel5;
              OD 1,0,0; ZEICHNE Atom;
  DREHE Z,60; OD 0,1,0; ZEICHNE Atom;
  DREHE Z,60; OD 0,0,1; ZEICHNE Atom;
  DREHE Z,60; OD 1,1,0; ZEICHNE Atom;
  DREHE Z,60; OD 0,1,1; ZEICHNE Atom;
  DREHE Z,60; OD 1,0,1; ZEICHNE Atom;
END.
```

Wird dieses Beispiel dargestellt, so erscheinen die einzelnen Atome wie in Abb. I (im farbigen Bildteil) eingefärbt. Die Angabe OD 1,0,0 bedeutet beispielsweise

„rot", `OD 1,1,0` entspricht „gelb". Die für die letzten drei Atome gewählten Farben entsprechen übrigens den Komplementärfarben zu Rot, Grün und Blau, sind also die Basis CMY des subtraktiven Farbmodells.

11.2 Lichtquellenmodelle

Zur Eingabe von Lichtquellen sind die vier von PHIGS PLUS definierten Typen vorgesehen. Eine ambiente Lichtquelle wird durch das Schlüsselwort `AMBIENTELQ` beschrieben, die anderen durch `GERICHTETELQ`, `PUNKTLQ` und `STRAHLER`. Anhand des Moleküls werden in diesem Abschnitt die Lichtquellen kurz im einzelnen betrachtet. Eine Bilderserie zum Zusammenwirken von Lichtquellentypen und Oberflächeneigenschaften finden Sie in Abb. III. Die genaue Syntax und alle Parameter werden in Anhang A erläutert.

Bislang war für das Molekül eine gerichtete Lichtquelle mit der Farbe $\begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$ definiert. Setzen Sie beispielsweise die Farbe der Lichtquelle auf Rot und beobachten die resultierende Lichtverteilung. Durch die Beleuchtung mit einer roten Lichtquelle wirkt die gesamte Szene rot, die einzelnen Atome des Moleküls wirken nur noch mit ihrem jeweiligen Rotanteil. Durch die Verwendung der gerichteten Lichtquelle reflektieren alle Atome das Licht in die gleiche Richtung.

Wird die gerichtete Lichtquelle durch eine ambiente Lichtquelle, z.B.

```
AMBIENTELQ 1,1,1;
```

ersetzt, dann erreicht Licht aus allen Richtungen das Molekül. Es wird gleichfalls in alle Richtungen wieder gestreut, so daß das Molekül flach und eher wie großes „Konfetti" wirkt. Der Parameter der ambienten Lichtquelle gibt die Farbe der Lichtquelle an.

Für eine punktförmige Lichtquelle muß nicht nur die Farbe und die Position angegeben werden. Mit Hilfe sogenannter Abklingkoeffizienten a und b wird die Intensität der Lichtquelle mit der Entfernung vom Objekt zusätzlich abgeschwächt. Ist I_L die Intensität der Lichtquelle, dann wird am Objekt die Intensität

$$I_O = \frac{1}{a + br^2} \cdot I_L$$

berücksichtigt. Im einfachsten Fall wird $a = 1$ und $b = 0$ gesetzt. Nehmen Sie für das Molekül beispielsweise:

```
PUNKTLQ 1,1,1 0,0,2 1,0;
```

Die Punktlichtquelle ist also in der Mitte über dem Molekül positioniert (Position $\begin{pmatrix} 0 & 0 & 2 \end{pmatrix}$) und die Intensität wird direkt ohne Abklingen übernommen ($a = 1, b = 0$). Sie erkennen deutlich den Unterschied zur gerichteten Lichtquelle von oben. Je größer Sie nun den zweiten Abklingkoeffizienten wählen, desto weniger Licht erreicht das Molekül. Für $a = 1$ und $b = 0.2$ ist die Intensität der

Lichtquelle schon deutlich zurückgegangen, für $a = 1$ und $b = 0.5$ kaum noch sichtbar.

Am umfangreichsten ist die Parameterliste für Spotlichtquellen. Neben der bereits bekannten Farbe, Position und den Abklingkoeffizienten muß zusätzlich noch mindestens die Lichtrichtung angegeben werden. Ersetzen Sie die Punktlichtquelle in unserem Beispiel durch einen Strahler mit der Position $\begin{pmatrix} 0 & 0 & 3 \end{pmatrix}$, der „nach unten" zeigt, also in negative z-Richtung:

```
STRAHLER 1,1,1 0,0,-1 0,0,3 1,0 1 90;
```

Die Lichtquelle sitzt jetzt etwas höher. Ihr Schattenkegel ist deutlich sichtbar auf dem Molekül zu erkennen. Wird die Strahlungsrichtung von $\begin{pmatrix} 0 & 0 & -1 \end{pmatrix}$ nun noch mehr in x-Richtung auf $\begin{pmatrix} 1 & 0 & -1 \end{pmatrix}$ verschoben, so werden die linken drei Kugeln nicht mehr vom Strahler erfaßt.

Modifizieren wir stattdessen den Öffnungswinkel, so verschwindet mit

```
STRAHLER 1,1,1 0,0,-1 0,0,3 1,0 1 180;
```

die scharfe Grenze des Lichtkegels, da der gesamte untere Halbraum beschienen wird. Das Bild wirkt wie mit einer Punktlichtquelle beleuchtet. Wird der Öffnungswinkel dagegen verengt, z.B. auf 70°, so werden die Atome deutlich weniger von der Lichtquelle beschienen. Mit Veränderung des Konzentrationsexponenten, z.B. auf den Wert 5, verdichtet sich die Beleuchtung in Richtung des – hier nicht sichtbaren – Ursprungs. Als Folge davon wird die Beleuchtung des Moleküls dunkler.

11.3 Oberflächenmodelle

Bis zu diesem Zeitpunkt haben wir stillschweigend davon Gebrauch gemacht, daß die Oberfläche der Atome alles Licht, ob ambient oder gerichtet, reflektieren. Natürlich können auch diese Parameter eingestellt werden. Ihre Vorbelegungswerte bedeuten tatsächlich, daß ambient, diffus und spiegelnd gleichermaßen reflektiert wird.

Oberflächeneigenschaften werden ebenso wie Transformationen *vor* den Objekten, die sie betreffen, angegeben. Dies hatten wir mit der Angabe `OD` bereits ausprobiert. Im Gegensatz zu Transformationen werden Sie allerdings nicht miteinander verknüpft, sondern bei Redefinition durch die neuere Beschreibung ersetzt.

Mit dem bisherigen Kenntnisstand können Sie nun mit weiteren Parametern experimentieren: dem ambienten, diffusen und spiegelnden Reflexionskoeffizienten `KA`, `KD` und `KS`, der diffusen und der spiegelnden Farbe `OD` und `OS`, dem Exponenten für die spiegelnde Reflexion `OE` und der Transparenz `T`. Weitere Oberflächeneigenschaften werden Sie in den nächsten Abschnitten noch kennenlernen. Für die Experimente wird ein neues Beispiel mit zwei „Atomen" eingeführt, wobei die Parameter des einen konstant bleiben und die des anderen verändert werden sollen:

```
GED_VERSION 1;

OBJEKT Atom;
  SKALIERE 2.0;
  KUGEL;
ENDOBJEKT;

GERICHTETELQ 1,1,1 1,1,-1;

SZENE Beispiel6;
  OD 1,0,0;
  VERSCHIEBE 2,0,0;
  ZEICHNE Atom;
  VERSCHIEBE -4,0,0;

  // An dieser Stelle werden die Parameter der folgenden
  // Beispiele gesetzt ...
  ZEICHNE Atom;
END.
```

Die Lichtquelle soll in diesem Beispiel ebenfalls unverändert bleiben.

Von einem Objekt wird Licht ambient, diffus und spiegelnd reflektiert. Die Koeffizienten KA, KD und KS geben an, welcher Teil des einfallenden Lichtes jeweils wie reflektiert wird. Die Vorbelegung dieser Werte ist KA = KD = KS = 1/3. Setzen Sie vor das zweite Atom der Testszene die Zeile:

```
KA 0; KD 1; KS 0;
```

so reflektiert die linke Kugel alles einfallende Licht diffus mit der Farbe des Objektes OD, hier also rot. Wird dagegen das gesamte Licht spiegelnd reflektiert

```
KA 0; KD 0; KS 1;
```

dann erscheint die Kugel *weiß*! Wie im Grundlagenkapitel erwähnt, reflektieren Objekte im diffusen und spiegelnden Bereich mit unterschiedlichen Farben. Dieser sehr realistische Ansatz – schauen Sie sich in Ihrem Zimmer um – wird durch die Angaben OD und OS gesteuert. Die vorbelegten Werte sind (siehe Tabelle A.1): $\texttt{OD} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$ und $\texttt{OS} = \begin{pmatrix} 1 & 1 & 1 \end{pmatrix}$. Da die rechte Kugel sowohl diffus als auch spiegelnd und ambient reflektiert, erscheint sie mit einer Mischung der Farben aus den letzten beiden Experimenten mit der linken Kugel.

Der ambiente Anteil wird nur bei Vorhandensein einer ambienten Lichtquelle aktiv. Setzen Sie also in diesem Beispiel mit nur einer gerichteten Lichtquelle die Objekteigenschaften mit der folgenden Zeile

```
KA 1; KD 0; KS 0;
```

so wird die linke Kugel nicht beleuchtet und bleibt daher schwarz.

Sinnvollerweise bleibt die Summe der drei Reflexionskoeffizienten KA, KD und KS kleiner oder gleich Eins. Wird sie größer, können Überbelichtungen der Szene auftreten.

Um die Wirkung des Helligkeitskoeffizienten beobachten zu können, kehren wir nochmals zur vorletzten Parameterwahl zurück. Setzen Sie für das linke und das rechte Atom die Werte

```
KA 0; KD 0; KS 1;
```

Das linke erhält den Exponenten `OE 20`, das rechte `OE 1`, den voreingestellten Wert. Bei der direkten Gegenüberstellung erkennen Sie, wie dieser Exponent die Größe des sogenannten Highlights beeinflußt. Je größer der Exponent, desto kleiner die Stelle der direkten Reflexion einer Lichtquelle. Sinnvolle Werte dieses Exponenten liegen zwischen 1 und etwa 200.

Mit einem letzten Parameter, `T`, beeinflussen Sie die Transparenz eines Objektes (siehe auch Abschnitt 13.5). Eine Transparenz von 0 bewirkt, daß ein Objekt undurchsichtig (opak) erscheint. Für durchsichtige Körper wird ein Wert zwischen 0 und 1 angegeben, der dem Prozentsatz entspricht, mit dem die Farbe des Objektes mit der Farbe des Hintergrundes gemischt wird. Vollständig transparente Körper mit `T` = 1.0 verschwinden „im Hintergrund".

Experimentieren Sie mit der Transparenz, indem Sie in unserer Beispieldatei beiden Kugeln die gleichen Parameter geben, der linken jedoch nacheinander Transparenzwerte von $0.0, 0.2, 0.5, 0.8$ und 1.0. Beobachten Sie, wie die rote Kugel mehr und mehr im blauen Hintergrund verschwindet.

12. Implementierung der Beleuchtungsmodelle

Beleuchtungsmodelle für das Rendering und für die Radiosity-Berechnung sind in der Pascal-Unit `primlight` implementiert. Innerhalb diese Moduls sind zwei Prozeduren definiert: `Beleuchte` berechnet aus der eingegebenen Geometrie und den Objekteigenschaften den diffus reflektierten und den spiegelnd reflektierten Farbanteil. Die Routinen `pow` und `pow1` berechnen die Exponentiation x^y für $0 \leq x \leq 1$ bzw. $0 \leq x < \infty$. Sie werden hier nicht näher beschrieben. Im Prinzip kann `primlight` auch für das Raytracing benutzt werden. Alle nötigen Informationen für dieses Verfahren sind vorhanden. Das Einbauen selbst überlassen wir Ihnen „zur Übung".

12.1 Datenstrukturen

Lokale Beleuchtungsmodelle, wie wir sie hier verwenden, basieren auf den folgenden Daten:

- der Geometrie an der Grenzfläche,
- den optischen Eigenschaften des Objektes und
- den Eigenschaften der Lichtquellen.

Dementsprechend wird die Prozedur `Beleuchte` durch die folgenden Parameter definiert:

```
PROCEDURE Beleuchte((* Eingabeparameter: *)
                    VAR p:           PunktTYP;
                    VAR Eigenschaft: EigenschaftTYP;
                    LichtQuellen:    LichtquellenPtrTYP;
                    VAR Auge:        Vektor3dTYP;
                    (* Ausgabeparameter: *)
                    VAR DiffFarbe,
                        SpekFarbe:   FarbVektorTYP);
```

Im Parameter `p` mit dem Typ `PunktTYP` wird unter anderem die `Koordinate` und die `Normale` an diesem Punkt übergeben.

```
PunktTYP   = RECORD
               Koordinate     : Vektor3DTYP;
               Normale        : Vektor3DTYP;
               ...
             END;
```

Die optischen Eigenschaften der Oberfläche werden im Parameter `Eigenschaft` übergeben. Sie umfassen die diffuse und spiegelnde Farbe (`OD` und `OS`), den Reflexionsexponenten `Oe` und den ambienten, diffusen und spiegelnden Reflexionsgrad (`Ka, Kd` und `Ks`). `EigenschaftTYP` ist eine Sammlung dieser Daten.

```
EigenschaftTYP      = RECORD
                        Ka,Kd,Ks,Oe,T,N   : SINGLE;
                        OD,OS             : FarbVektorTYP;
                        ...
                      END;
```

Die Lichtquellen mit ihren geometrischen und optischen Eigenschaften werden in der verketteten Liste `Lichtquellen` übergeben. Da die Lichtquellen je nach Typ durch verschiedene Parameter definiert sind, wird für `LichtquellenTYP` eine variante Datenstruktur verwendet. Wir lassen in Analogie zu PHIGS PLUS vier Lichtquellentypen zu: ambiente, Richtungs-, Punkt- und Strahlerlichtquellen. Die Parameter wurden bereits erläutert, so daß wir uns hier auf die Darstellung der Typen beschränken.

```
AmbienteLqTYP       = RECORD
                        Farbe           : FarbVektorTYP;
                      END;
GerichteteLqTYP     = RECORD
                        Farbe           : FarbVektorTYP;
                        Richtung        : Vektor3DTYP;
                      END;
PunktLqTYP          = RECORD
                        Farbe           : FarbVektorTYP;
                        Position        : Punkt3DTYP;
                        AbklingKoeff    : Vektor2D;
                      END;
StrahlerTYP         = RECORD
                        Farbe           : FarbVektorTYP;
                        Richtung        : Vektor3DTYP;
                        Position        : Punkt3DTYP;
                        AbklingKoeff    : Vektor2D;
                        Oe              : SINGLE;
                        OeffnungsWinkel : SINGLE;
                      END;
```

Die verkettete Liste wird mit `LichtquellenTYP` erzeugt:

```
LichtquellenTYP =
   RECORD
     next         : LichtquellenPtrTYP;
     nichtVerdeckt: BOOLEAN;

     CASE Art: ArtDerLqTYP OF
       AmbienteLq   : (AmbienteLq  : AmbienteLqTYP);
       GerichteteLq : (GerichteteLq: GerichteteLqTYP);
       PunktLq      : (PunktLq     : PunktLqTYP);
       Strahler     : (Strahler    : StrahlerTYP);
     END;
   END;
```

Mit `NichtVerdeckt` kann eine Lichtquelle „ein"- und „ausgeschaltet" oder bei der Strahlverfolgung als verdeckt markiert werden.

Der letzte Eingabeparameter ist die Position der „virtuellen Kamera" `Auge`. Alle Rechnungen werden in Weltkoordinaten durchgeführt. `DiffFarbe` und `SpekFarbe` werden von `Beleuchte` zurückgegeben. Dabei ist `DiffFarbe` noch nicht in die diffuse Reflektivität des zu beleuchtenden Punktes eingerechnet. Der Grund wurde bereits in Abschnitt 9.3 erläutert: Ein Punkt kann Eckpunkt verschiedener Flächen mit verschiedenen diffusen Farben sein. `FarbVektorTYP` beschreibt in unserer Implementierung das Farbtripel aus Rot-, Grün- und Blauanteil.

```
TYPE _3TYP          = RECORD CASE INTEGER OF
                        1:  ...
                        2:  ...
                        3:  (R, G, B: SINGLE);
                      END;

     FarbVektorTYP  = _3TYP;
```

12.2 Algorithmen

Das Verfahren zur Berechnung der Beleuchtung besteht im wesentlichen aus Initialisierungen und einer Schleife über alle Lichtquellen. In der Schleife werden die einzelnen Beiträge der Lichtquellen zur diffusen und spiegelnden Farbe aufaddiert.

```
PROCEDURE Beleuchte(...);

VAR ...
    ve: Vektor3DTYP;          (* Einheitsvektor von p zum Auge   *)
    ...
    lptr: LichtQuellenPtrTYP; (* Zeiger auf aktuelle Lichtquelle *)

BEGIN
  ve.x := auge.x - p.Koordinate.x;     (* Vektor von p zum Auge *)
```

```
    ve.y := auge.y - p.Koordinate.y;
    ve.z := auge.z - p.Koordinate.z;
    NormiereVektor(ve);
    lptr:=Lichtquellen;
    DiffFarbe:=schwarz;
    SpekFarbe:=schwarz;
    WHILE lptr<>NIL DO BEGIN
      ...

      lptr:=lptr^.next;
    END;
  END;
```

Die Initialisierung besteht aus der Berechnung und Normierung des Blickvektors `ve` sowie der Initialisierung der diffusen und der spiegelnden Farbe. Da das RGB-Modell ein additives Farbmodell ist, werden `DiffFarbe` und `SpekFarbe` mit `schwarz` vorbelegt. `schwarz` ist die in der globalen Unit `primglob` bestimmte Konstante `(R:0; G:0; B:0)`.

Anhand der Variablen `lptr^.Art` werden die Lichtquellen unterschieden. Ambiente Lichtquellen beeinflussen nur den diffusen Farbanteil, spiegelnde Reflexion tritt nicht auf.

```
...
AmbienteLq: BEGIN
  DiffFarbe.r:=DiffFarbe.r+
     lptr^.AmbienteLq.Farbe.r*Eigenschaft.Ka;
  DiffFarbe.g:=DiffFarbe.g+
     lptr^.AmbienteLq.Farbe.g*Eigenschaft.Ka;
  DiffFarbe.b:=DiffFarbe.b+
     lptr^.AmbienteLq.Farbe.b*Eigenschaft.Ka;
END;
...
```

Für den diffusen Anteil bei Richtungslichtquellen muß zuerst das Skalarprodukt aus Flächennormale und Lichtquellenvektor berechnet werden. Es wird in `cosa` gespeichert, um bei der Berechnung der spiegelnden Reflexion wiederverwendet zu werden. Der diffuse Anteil ist mit dem Modell von Gouraud berechnet. Anschließend wird für den spiegelnden Anteil der reflektierte Lichtquellenvektor bestimmt. Die spiegelnde Reflexion wird mit dem Beleuchtungsmodell von Phong inkrementiert.

```
...
GerichteteLq: BEGIN
  cosa:=p.Normale.x * (-lptr^.GerichteteLq.Richtung.x) +
        p.Normale.y * (-lptr^.GerichteteLq.Richtung.y) +
        p.Normale.z * (-lptr^.GerichteteLq.Richtung.z);
  IF cosa>0.0 THEN BEGIN       (* Lichtquelle von vorne?   *)
    DiffFarbe.r:=DiffFarbe.r+lptr^.GerichteteLq.Farbe.r*
                    Eigenschaft.Kd*cosa;
```

```
    DiffFarbe.g:=DiffFarbe.g+lptr^.GerichteteLq.Farbe.g*
                    Eigenschaft.Kd*cosa;
    DiffFarbe.b:=DiffFarbe.b+lptr^.GerichteteLq.Farbe.b*
                    Eigenschaft.Kd*cosa;
    tcosa:=2.0*cosa;          (* fuer Reflexionsvektor     *)
    vr.x := tcosa*p.Normale.x + lptr^.GerichteteLq.Richtung.x;
    vr.y := tcosa*p.Normale.y + lptr^.GerichteteLq.Richtung.y;
    vr.z := tcosa*p.Normale.z + lptr^.GerichteteLq.Richtung.z;
    cosb := ve.x * vr.x + ve.y * vr.y + ve.z * vr.z ;
    IF cosb > 0.0 THEN BEGIN   (* Reflexionsvektor sichtbar? *)
      fl:=pow(cosb,Eigenschaft.Oe);
      SpekFarbe.r:=SpekFarbe.r+lptr^.GerichteteLq.Farbe.r*
                        Eigenschaft.Ks*Eigenschaft.Os.r*fl;
      SpekFarbe.g:=SpekFarbe.g+lptr^.GerichteteLq.Farbe.g*
                        Eigenschaft.Ks*Eigenschaft.Os.g*fl;
      SpekFarbe.b:=SpekFarbe.b+lptr^.GerichteteLq.Farbe.b*
                        Eigenschaft.Ks*Eigenschaft.Os.b*fl;
    END;
  END;
END;
...
```

Punktlichtquellen erfordern zusätzlich die Berechnung des Lichtquellenvektors `vl`. Die Abschwächung des Lichtes durch die Entfernung ist in `La` mit Hilfe zweier Abklingkoeffizienten modelliert. Die restliche Berechnung ist analog zu der für Richtungslichtquellen.

```
...
PunktLq: BEGIN
  vl.x := lptr^.PunktLq.Position.x - p.Koordinate.x;
  vl.y := lptr^.PunktLq.Position.y - p.Koordinate.y;
  vl.z := lptr^.PunktLq.Position.z - p.Koordinate.z;
  norm := sqrt(sqr(vl.x)+sqr(vl.y)+sqr(vl.z));
  La := 1.0 / (lptr^.PunktLq.AbklingKoeff[0] +
              lptr^.PunktLq.AbklingKoeff[1] * norm*norm );
  IF La>0 THEN BEGIN
    IF abs(norm)<1E-10 THEN
       Fehler('Beleuchte: Objekt trifft Lichtquelle');
    vl.x:=vl.x/norm; vl.y:=vl.y/norm; vl.z:=vl.z/norm;
    cosa :=p.Normale.x*vl.x+p.Normale.y*vl.y+p.Normale.z*vl.z;
    IF cosa>0.0 THEN BEGIN       (* Lichtquelle vor p?  *)
      fl:=cosa*La; tcosa:=2.0*cosa;
      DiffFarbe.r:=DiffFarbe.r+lptr^.PunktLq.Farbe.r*
                        Eigenschaft.Kd*fl;
      DiffFarbe.g:=DiffFarbe.g+lptr^.PunktLq.Farbe.g*
                        Eigenschaft.Kd*fl;
      DiffFarbe.b:=DiffFarbe.b+lptr^.PunktLq.Farbe.b*
                        Eigenschaft.Kd*fl;
      tcosa:=2.0*cosa;           (* fuer Reflexionsvektor     *)
```

```
      vr.x := tcosa*p.Normale.x -vl.x;
      vr.y := tcosa*p.Normale.y -vl.y;
      vr.z := tcosa*p.Normale.z -vl.z;
      cosb := ve.x * vr.x + ve.y * vr.y + ve.z * vr.z ;
      IF cosb > 0.0 THEN BEGIN   (* Reflexionsvektor sichtbar? *)
        fl:=pow(cosb,Eigenschaft.Oe)*La;
        SpekFarbe.r:=SpekFarbe.r+lptr^.PunktLq.Farbe.r*
                          Eigenschaft.Ks*Eigenschaft.Os.r*fl;
        SpekFarbe.g:=SpekFarbe.g+lptr^.PunktLq.Farbe.g*
                          Eigenschaft.Ks*Eigenschaft.Os.g*fl;
        SpekFarbe.b:=SpekFarbe.b+lptr^.PunktLq.Farbe.b*
                          Eigenschaft.Ks*Eigenschaft.Os.b*fl;
      END;
    END;
  END;
END;
...
```

Der eingeschränkte Strahlungsbereich von Spotlichtquellen muß bei der Beleuchtung berücksichtigt werden. Ist der Lichtquellenvektor innerhalb des Öffnungswinkels um die Hauptstrahlungsrichtung `Strahler.Richtung`, dann werden diffuser und spiegelnder Farbanteil berechnet.

```
...
Strahler: BEGIN
  vl.x := lptr^.Strahler.Position.x - p.Koordinate.x;
  vl.y := lptr^.Strahler.Position.y - p.Koordinate.y;
  vl.z := lptr^.Strahler.Position.z - p.Koordinate.z;
  norm := sqrt(sqr(vl.x)+sqr(vl.y)+sqr(vl.z));
  IF abs(norm)<1E-10 THEN
     Fehler('Beleuchte: Objekt trifft Lichtquelle');
  vl.x:=vl.x/norm; vl.y:=vl.y/norm; vl.z:=vl.z/norm;
  cosc := - lptr^.Strahler.Richtung.x * vl.x
          - lptr^.Strahler.Richtung.y * vl.y
          - lptr^.Strahler.Richtung.z * vl.z;
  IF cosc > cos(lptr^.Strahler.OeffnungsWinkel*0.5) THEN BEGIN
    La := 1.0 / (lptr^.Strahler.AbklingKoeff[0] +
                 lptr^.Strahler.AbklingKoeff[1] * norm*norm );
    IF La>0 THEN BEGIN
      cosa :=p.Normale.x*vl.x+p.Normale.y*vl.y+p.Normale.z*vl.z;
      fl1:=pow1(cosc,lptr^.Strahler.Oe)*La;
      fl:= cosa*fl1; tcosa:=2.0*cosa;
      IF cosa>0.0 THEN BEGIN       (* Lichtquelle vor p?  *)
        fl:=cosa*La; tcosa:=2.0*cosa;
        DiffFarbe.r:=DiffFarbe.r+lptr^.Strahler.Farbe.r*
                          Eigenschaft.Kd*fl;
        DiffFarbe.g:=DiffFarbe.g+lptr^.Strahler.Farbe.g*
                          Eigenschaft.Kd*fl;
        DiffFarbe.b:=DiffFarbe.b+lptr^.Strahler.Farbe.b*
```

```
                        Eigenschaft.Kd*fl;
          tcosa:=2.0*cosa;           (* fuer Reflexionsvektor *)
          vr.x := tcosa*p.Normale.x -vl.x;
          vr.y := tcosa*p.Normale.y -vl.y;
          vr.z := tcosa*p.Normale.z -vl.z;
          cosb := ve.x * vr.x + ve.y * vr.y + ve.z * vr.z ;
          IF cosb > 0.0 THEN BEGIN   (* Reflexionsvektor sichtbar? *)
            fl:=pow1(cosb,Eigenschaft.Oe)*fl1;
            SpekFarbe.r:=SpekFarbe.r+lptr^.Strahler.Farbe.r*
                        Eigenschaft.Ks*Eigenschaft.Os.r*fl;
            SpekFarbe.g:=SpekFarbe.g+lptr^.Strahler.Farbe.g*
                        Eigenschaft.Ks*Eigenschaft.Os.g*fl;
            SpekFarbe.b:=SpekFarbe.b+lptr^.Strahler.Farbe.b*
                        Eigenschaft.Ks*Eigenschaft.Os.b*fl;
          END;
        END;
      END;
    END;
  END;
  ...
```

Sollen neue Lichtquellentypen zu den bereits vorhandenen dazugefügt werden, muß `LichtquellenTYP` um diesen neuen Typ ergänzt und `Beleuchte` um einen weiteren Fall erweitert werden.

13. Darstellungsmethoden (Rendering)

Nachdem bislang die Modellbildung bei der Darstellung von Szenen im Vordergrund stand, liegt im folgenden der Schwerpunkt auf den algorithmischen Aspekten. Dazu stellen wir ein Verfahren genauer vor, mit dem die Verdeckungsrechnung gelöst werden kann: das z-Buffer-Verfahren. Anschließend wird die Umsetzung von Dreiecken und deren Verallgemeinerungen, den Polygonen, in die vom Bildschirm darstellbare Form besprochen. Diese Verfahren erweitern wir um die Farbdarstellung. Am Schluß des Kapitels folgen weitere Methoden, um die Realitätsnähe in einem Bild noch zu steigern.

13.1 Verdeckungsrechnung mit dem z-Buffer-Verfahren

Eine der einfachsten Möglichkeiten, das Verdeckungsproblem zu lösen, ist der pixelweise Vergleich der z-Werte der Objekte. Zusätzlich zum Farbwert (frame buffer) wird für jedes Pixel auch noch der Tiefenwert z gespeichert. Der Aufwand an Speicherplatz für dieses Verfahren ist relativ hoch, dafür ist der Algorithmus einfach.

Alle Objekte sind für die Darstellung in normierte Koordinaten transformiert. Diese bewegen sich für den z-Wert zwischen 0 und 1, wobei $z = 0$ „vorne“ und $z = 1$ „hinten“ entspricht. Die Initialisierung des z-Buffers belegt alle Speicherzellen mit dem Wert 1. Bei der Verwendung von ganzen Zahlen für den z-Buffer wird der Bereich $[0, 1]$ z.B. auf $0, \ldots, 32767$ abgebildet. Dieser Wert entspricht dem z-Wert des Hintergrundes. Der Bildschirmspeicher wird mit der Farbe des Hintergrundes gesetzt. Die Objekte werden nun nacheinander behandelt und mit einem Rasterisierungsverfahren in den Speicher geschrieben.

Ist der für einen Punkt des Objektes bestimmte z-Wert niedriger als der Wert, der bereits gespeichert ist, dann wird die Farbe des Objektes in den Bildspeicher geschrieben. Das Objekt liegt also an dieser Stelle vor dem bereits gespeicherten. Der neue z-Wert wird in den z-Buffer geschrieben. Ist der z-Wert dagegen größer als der im z-Buffer vorhandene, dann ist das Objekt an dieser Stelle verdeckt und die Werte werden nicht eingetragen.

Algorithmus 13.1 (z-Buffer)

FÜR alle Punkte des Bildschirms:
Initialisiere den Farbspeicher mit der Hintergrundfarbe.
Initialisiere den z-Buffer mit dem am weitesten entfernten Wert.
ENDE FÜR alle Punkte des Bildschirms.
FÜR alle Objekte (Polygone/Dreiecke) der Szene:
Bestimme den z-Wert für Pixel (x, y) des Objektes.
WENN z näher als der gespeicherte z-Wert ist, DANN:
Ersetze den gespeicherten z-Wert durch den neuen z-Wert.
Ersetze den Farbwert durch den neuen Farbwert.
ENDE WENN z näher.
ENDE FÜR alle Objekte.

Um den Speicherplatz für den z-Buffer in Grenzen zu halten, bewegt der z-Wert sich in einem Bereich von 0..`max`. `max` ist oft 2^{16}, in teureren Systemen auch 2^{24}. Die Diskretisierung des z-Wertes kann Fehler in der Darstellung zur Folge haben, wenn die Entscheidung über „verdeckt" oder „nicht verdeckt" falsch getroffen wird. Daher werden in Weltkoordinaten vor der Transformation in das normierte Koordinatensystem zwei Ebenen eingeführt, die den Sichtbereich nach vorne und hinten begrenzen. Diese als vordere und hintere Clipping-Ebene bezeichneten Flächen helfen also bei sinnvoller Wahl, die Fehler im z-Buffer-Verfahren zu verringern.

13.2 Scankonvertierung von Dreiecken und Polygonen

Im Grundlagenkapitel wurde bereits die Umsetzung von Linien auf ein Rastergerät beschrieben. Um eine flächige Darstellung der gedachten Objekte zu erreichen, müssen wir nun überlegen, wie Dreiecke, oder allgemein planare n-Ecke, auf das Raster abgebildet werden können. Dabei soll das z-Buffer-Verfahren für die Verdeckungsrechnung eingesetzt werden. Das Füllen der Flächen mit Farbe wird später ergänzt.

Die Aufgabe, die bei der Scankonvertierung zu lösen ist, besteht in der Bestimmung aller x- und y-Koordinaten auf dem Bildschirm, die von einem Polygon bedeckt werden. An dieser Stelle muß der z-Wert bestimmt werden. Im Prinzip wird die Scankonvertierung von Polygonen auf die Untersuchung einzelner Rasterzeilen und anschließend auf die einzelner Punkte zurückgeführt. Das bedeutet, daß für alle konstanten y alle x gesucht werden, an denen ein Polygon die Scanlinie schneidet. Anschließend wird die Liste der erhaltenen Punkte sortiert, so daß eine Reihe von Scanlinienstücken übrigbleibt, deren Darstellung dann das Polygon ergibt (Abb. 13.1).

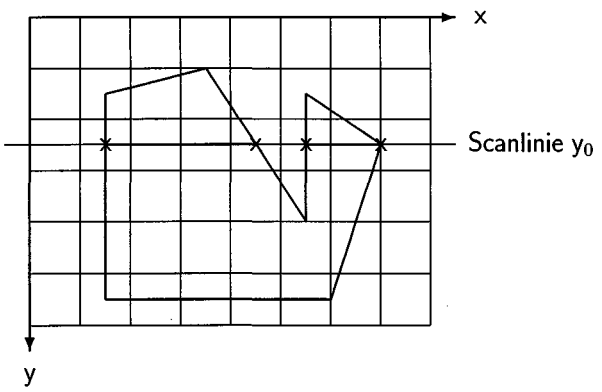


Abb. 13.1 Scankonvertierung von Polygonen

Eine Vereinfachung des Algorithmus tritt ein, wenn Dreiecke betrachtet werden. Sie sind auf alle Fälle planar, einfach zu berechnen und darzustellen. Jedes Polygon kann in Dreiecke zerlegt werden. Somit läßt sich die Aufgabe, ein Polygon darzustellen, auf die Aufgabe zurückführen, ein Dreieck darzustellen.

Sei ein Dreieck durch die Punkte P_1, P_2 und P_3 gegeben. Die Spezialfälle, daß ein Dreieck zum Punkt ($P_1 = P_2 = P_3$) oder zu einer Linie degeneriert, sollen vorerst ausgeschlossen werden. Dann können die drei Punkte so nach y sortiert werden, daß $P_{1y} \leq P_{2y} \leq P_{3y}$ gilt. Das Dreieck ist also durch P_2 in zwei Teildreiecke aufgeteilt, die nacheinander betrachtet werden. Mit Hilfe der Interpolation an den Kanten $\overline{P_1P_2}$ und $\overline{P_1P_3}$ werden für jede Zeile y der linke und der rechte x-Wert bestimmt. Wegen der Linearität von z über die Fläche kann mit dz/dx und dz/dy parallel dazu der z-Wert jedes Pixels berechnet werden. Nach Erreichen des Punktes P_2 werden die Kanten $\overline{P_1P_3}$ und $\overline{P_2P_3}$ betrachtet. Eine genauere Beschreibung folgt im Implementierungsteil.

13.3 Linear interpolierende Schattierungsverfahren

In Verbindung mit der Scankonvertierung kann auch die Schattierungsrechnung ausgeführt werden, wenn linear interpolierende Verfahren verwendet werden. Im Gegensatz zur Strahlverfolgung oder zur Strahlungsberechnung nutzen sie lokale Beleuchtungsmodelle. Der Wunsch bei der Schattierungsrechnung ist, mit möglichst geringem Aufwand einen relativ guten Eindruck einer Beleuchtungssituation zu vermitteln.

Die einfachste Art, ein Objekt flächig darzustellen, ist sicher, diese in der ihm gegebenen Farbe (der diffusen Reflektivität) darzustellen. Das Objekt erscheint als Silhouette, da alle Flächen gleich hell erscheinen. Das hinterläßt natürlich keinen allzu realistischen Eindruck.

Der einfachste Algorithmus zur Schattierung von Polygonen belegt ein Polygon gleichmäßig mit einer Farbe. Diese Farbe wird z.B. am Mittelpunkt der Fläche mit einem Beleuchtungsmodell berechnet und anschließend als konstantes Attribut der Fläche gespeichert. Dieser Algorithmus heißt konstante Schattierung, in englisch „flat shading".

Das Resultat dieses Algorithmus ist allerdings noch nicht sehr realistisch (siehe auch Abb. II im farbigen Bildteil). Die Verwendung von Beleuchtungsmodellen für die spiegelnde Reflexion ist nicht allzu sinnvoll, da keine Helligkeitsänderungen über einer Fläche dargestellt werden kann. Die Flächenmodelle stellen eine Annäherung an gekrümmte Objekte dar, so daß mit dem „flat shading" dargestellte Objekte immer etwas „eckig" erscheinen. Ist die Annäherung allerdings so fein, daß die einzelnen Flächen nurmehr wenige Pixel belegen, dann kann auch eine konstante Schattierung ausreichend sein.

Ein Weg, die Nachteile des Flat-Shading zu beseitigen, ist die lineare Interpolation von Farbwerten. Dieses Verfahren heißt „Gouraud-Shading" und ist nach Henri Gouraud benannt, der es erstmals 1971 veröffentlicht hat [Gou71]. Die Idee ist, das Beleuchtungsmodell an den Eckpunkten eines Polygons zu berechnen und anschließend die Farbe über die Fläche zu interpolieren. Handelt es sich dabei um ein Dreieck, so ist der Farbverlauf analog zum Verlauf der z-Koordinate linear über der Fläche. Es kann also für die Berechnung auf die gleichen Parameter wie für die z-Interpolation zurückgegriffen werden.

Das bedeutet, daß analog zu den Steigungen dz/dx und dz/dy für das Dreieck konstante Farbsteigungen dc/dx und dc/dy gelten. Diese werden in jedem Schritt, in dem sich x bzw. y ändert, entsprechend akkumuliert.

Der Normalenvektor für die Berechung des Beleuchtungsmodells kann auf verschiedene Art und Weise berechnet werden. Im allgemeinen hängen an einem Punkt mehrere Flächen. Jede dieser Flächen hat, da sie planar ist, einen Normalenvektor. Soll die Farbe in einem weichen Übergang zwischen den Flächen verteilt werden, so muß der Normalenvektor im Punkt als Mittel der angrenzenden Flächennormalen berechnet werden. Ist dagegen gewollt, daß der Punkt oder eine bestimmte Kante kantig erscheinen, dann muß das Beleuchtungsmodell mehrmals mit den verschiedenen Flächennormalen berechnet werden. Solche Punkte werden in der Eingabesprache als „scharf" bezeichnet.

Die Ergebnisse des Verfahrens sind dann befriedigend, wenn nur diffuse Reflexion betrachtet wird und die Annäherung der Polygone an das Objekt fein genug ist (Abb. II). Bei der Darstellung spiegelnder Körper können die heller erscheinenden Stellen verschoben und in der Form nicht korrekt wiedergegeben sein. Weiterhin können sogenannte „Mach-Band-Effekte" auftreten. Diese werden wahrgenommen, wenn ein Farbverlauf zwar stetig, nicht aber stetig differenzierbar ist. Das Auge verstärkt die Unstetigkeit, so daß sie deutlicher sichtbar erscheint (Abb. 13.2).

Der nächste Schritt zur Verbesserung der Farbverläufe wurde von Bui Tong Phong 1975 eingeführt [Pho75]. Er schlägt vor, statt der Farben die Normalenvektoren über die Fläche zu interpolieren. Handelt es sich bei der Fläche wieder um ein Dreieck, dann kann die Interpolation gemeinsam mit der z-Interpolation

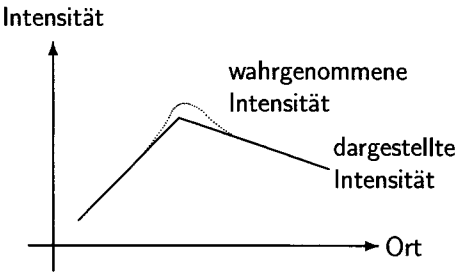


Abb. 13.2 Verstärkung von Unstetigkeiten durch den Mach-Band-Effekt

durchgeführt werden. Allerdings muß nun das Beleuchtungsmodell für jedes Pixel ausgeführt werden, so daß der Algorithmus wesentlich aufwendiger ist als der Gouraud-Algorithmus.

Die Qualität der mit dem Phong-Algorithmus erzeugten Bilder ist im allgemeinen besser als mit den vorher beschriebenen Algorithmen (Abb. II und XI). Insbesondere sind die hell reflektierenden Stellen durch die Interpolation der Normalen in ihrem Erscheinungsbild besser wiedergegeben.

Bei den bisher vorgestellten Verfahren fällt auf, daß sie sich in der Qualität und in der Rechenzeit voneinander unterscheiden. Auch die Fähigkeit, bestimmte Effekte wiedergeben zu können, z.B. die „highlights", sind unterschiedlich. Daraus entwickelte sich der Wunsch, die Verfahren genau dann einzusetzen, wenn dies „lohnenswert" ist. Diese Algorithmen werden unter dem Begriff „adaptive Verfahren" zusammengefaßt.

Eines dieser Verfahren stammt von Bergman und anderen [BFGS86]. In ihm werden grundsätzlich alle Polygone konstant beleuchtet. Überschreitet die Differenz der Intensitäten aneinandergrenzender Polygone eine bestimmte Schwelle, so werden sie mit einem Gouraud-Algorithmus interpoliert. Weicht schließlich die Differenz der Normalenvektoren um mehr als eine zulässigen Betrag voneinander ab, so wird zusätzlich eine Phong-Interpolation durchgeführt.

Aufbauend auf der Erkenntnis, daß einfachere Schattierungsverfahren eingesetzt werden können, wenn die Triangulierung eines Objektes nur fein genug gewählt ist, agiert ein weiteres Verfahren. Im sogenannten „highlight shading" von Hornung und Pöpsel werden Dreiecke in vier Unterdreiecke aufgeteilt, wenn die Differenz der Ergebnisse zwischen Phong- und Gouraud-Schattierung zu groß wird [PH90]. Das Ergebnis ist ein adaptiver Algorithmus, der relativ schnell ist und dabei eine dem Phong-Verfahren ähnliche Qualität erzeugt.

13.4 Texturen

Neben der Möglichkeit der Beleuchtung und Schattierung mit den vorgestellten Verfahren gibt es weitere Verfahren, um Bilder noch realistischer erscheinen zu lassen. Eines davon ist der Einsatz von Texturen.

Texturen sind umgangssprachlich gesehen Oberflächenstrukturen, beispielsweise die Struktur einer Tapete oder eines Teppichs, aber auch die farbliche Struktur von Marmor. In der generativen Computergrafik werden Texturen mit verschiedenen Verfahren eingesetzt. Es werden zweidimensionale und dreidimensionale (solid) Texturen unterschieden. Zweidimensionale Texturen, als Feld mit definierten Texturkoordinaten abgespeichert, können in einem Beleuchtungsmodell z.B. die Farbe oder den Normalenvektor eines Objektes modulieren. Ein Beispiel dafür ist in Abb. II im farbigen Bildteil zu sehen. Bei der Verwendung dieser Texturen erhält man die Vorstellung, daß dieses zweidimensionale Feld auf die Oberfläche eines Objektes „aufgeklebt" wird.

Dreidimensionale Texturen dagegen definieren ein Feld von Werten, aus dem ein Objekt quasi „ausgeschnitten" wird. Egal, um welche Art von Textur es sich handelt, eines der mit der Berechnung verbundenen Probleme ist die Bestimmung des Texturwertes für einen bestimmten Punkt auf der Oberfläche des Objektes. Es wird daher ein zusätzliches Koordinatensystem, das Texturkoordinatensystem, eingeführt. Von diesem besteht dann eine Abbildung in die Weltkoordinaten der Szene. Dies kann beispielsweise durch Zuordnung der Texturkoordinaten zu den Eckpunkten von Polygonen geschehen.

Bei der Darstellung werden die Texturkoordinaten dann – analog zu den Steigungen von z und der Farbe – interpoliert, um mit dem interpolierten Koordinatenwert aus der Textur den benötigten Wert auslesen zu können. Wegen der oftmals perspektivischen Transformation der Objekte ist eine lineare Interpolation der Texturkoordinaten allerdings nur eine Näherung, deren schlechte Qualität sich durch entsprechende Verzerrungen zeigt. Für eine genauere Texturaufbringung kann ebenfalls ein adaptives Verfahren gefunden werden, das eine Unterteilung der Polygone in Abhängigkeit von der Verzerrung der Textur vornimmt.

In den folgenden drei Abschnitten besprechen wir drei Texturarten näher.

13.4.1 Farbtextur-Mapping

Hierbei handelt es sich um die Abbildung eines zweidimensional definierten Feldes von Farbwerten auf die Oberfläche von Objekten. Im allgemeinen wird die Textur mit lokalen Koordinaten u und v versehen, die zwischen 0 und 1 laufen. Eine Oberfläche, die texturiert werden soll, muß daher in u und v parametrisiert werden (Abb. 13.3).

Für Ebenen und daher auch für Polygone ist dies leicht möglich mit der Parameterdarstellung.

$$P(u,v) = P_1 + uP_u + vP_v,$$

wobei

$$\begin{aligned} P_u &= \frac{1}{N}((v_3 - v_1)(P_2 - P_1) - (u_3 - u_1)(P_3 - P_1)) \\ P_v &= \frac{1}{N}(-(v_2 - v_1)(P_2 - P_1) + (u_2 - u_1)(P_3 - P_1)) \end{aligned}$$

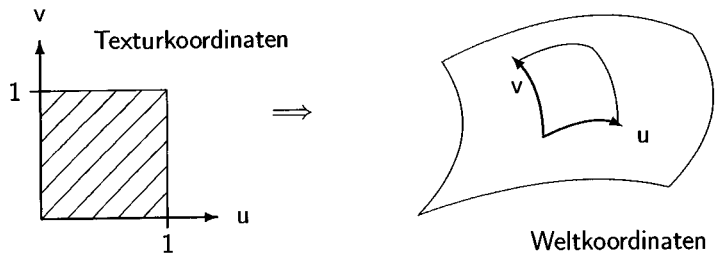


Abb. 13.3 Aufbringen von zweidimensionalen Texturen

mit

$$N = (u_2 - u_1)(v_3 - v_1) - (u_3 - u_1)(v_2 - v_1),$$

wenn P_1, P_2 und P_3 die Ebene und $(u_1, v_1), (u_2, v_2)$ und (u_3, v_3) die Parameterwerte an den Punkten definieren. In der praktischen Anwendung kann diese Abbildung bzw. deren Umkehrabbildung durch lineare Interpolation der Texturkoordinaten realisiert werden. Insbesondere bei perspektivischen Abbildungen werden dabei allerdings die Texturen verzerrt.

Weitere Probleme ergeben sich aus der Endlichkeit und der Diskretisierung der Textur. Die Endlichkeit kann durch periodisches Wiederholen der Textur, also entsprechende Modulo-Umrechnung von u und v, überwunden werden. Die Diskretisierung bekommt man durch Filterung in den Griff. Der aus der Textur ausgelesene Farbwert moduliert die diffuse Farbe eines Objektes.

13.4.2 Bumpmapping

Eine weitere, weit verbreitete Texturierungsart ist das sogenannte Bumpmapping. Bei diesem Verfahren wird nicht die Farbe mit einem Farbwert moduliert, sondern der Normalenvektor „gestört". Diese Störung erzeugt sichtbare Dellen und Beulen, daher der Name „bump" (Abb. IV).

Die Änderung des Normalenvektors muß vor der Berechnung des Beleuchtungsmodells stehen. Sie ist sonst unwirksam. Aus diesem Grund wird dieses Verfahren nur zusammen mit der Phong-Interpolation angewendet.

Die Grundidee besteht darin, mit einer Störungsfunktion $F(u, v)$ die Fläche in Richtung des Normalenvektors zu verfälschen.

$$P'(u, v) = P(u, v) + F(u, v)\mathbf{N}(u, v)$$

Die Tangentialvektoren P_u und P_v müssen entsprechend modifiziert werden,

$$P'_u = \frac{\partial P'(u, v)}{\partial u}, \qquad P'_v = \frac{\partial P'(u, v)}{\partial v}$$

ebenso wie der Normalenvektor, der dem Kreuzprodukt aus P'_u und P'_v entspricht.

$$\mathbf{N}' = P'_u \times P'_v$$

Nach Umformungen und Vereinfachungen kommt man zu einer Form

$$\mathbf{N}' = \mathbf{N} + QP_u + RP_v$$

mit

$$\begin{aligned} Q &= (F_v(P_u \cdot P_v) - F_u(P_v \cdot P_v))/|\mathbf{N}| \\ R &= (F_u(P_u \cdot P_v) - F_v(P_u \cdot P_u))/|\mathbf{N}|. \end{aligned}$$

Da diese Terme extrem rechenaufwendig sind, wird vereinfachend auch

$$\mathbf{N}' = \mathbf{N} + \Delta u P_u + \Delta v P_v$$

angenommen. Die „bump" -Textur besteht also aus den beiden Größen Δu und Δv, beides skalare Felder. Sie können aus der gespeicherten Textur durch Differenzbildung zweier benachbarter Texturwerte gewonnen werden (siehe auch die Erläuterungen zur Implementierung, Anhang A).

13.4.3 Volumentexturen

Dreidimensionale Texturen werden als „solid" bezeichnet. Sie sind in einem Texturkoordinatensystem definiert, aus dem die Objekte modelliert werden. Im einfachsten Fall entsprechen also die Modellkoordinaten den Texturkoordinaten. Das Problem bei dieser Texturart ist nicht so sehr die Transformation zwischen Modell- und Texturkoordinaten. Es ist vielmehr die Erzeugung realistisch wirkender Texturen.

Eine der bekanntesten Methoden zur Generierung solcher Texturen stammt von Ken Perlin [Per85,CP93]. Farbwerte werden beispielsweise mit

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} NoiseR(x,y,z) \\ NoiseG(x,y,z) \\ NoiseB(x,y,z) \end{pmatrix}$$

erzeugt. Die Funktionen $NoiseX$ müssen dabei mehrere Bedingungen erfüllen:

- Sie sind auf den Bereich zwischen 0 und 1 normiert,
- sie sind unter bestimmten Transformationen invariant und
- sie sind bandbegrenzt.

Diese Eigenschaften haben zur Folge, daß die Funktion sich nicht zu stark innerhalb einer bestimmten Umgebung ändert. $NoiseX$ wird mit Hilfe eines dreidimensionalen Feldes $\begin{pmatrix} x & y & z \end{pmatrix}$, in dem pseudozufällige Zahlen gespeichert werden, erzeugt.

13.5 Transparenz

Transparenz ist ein Effekt, der im Zusammenhang mit Beleuchtungsmodellen bereits diskutiert wurde. Im allgemeinen ist die Berechnung diese Effektes sehr aufwendig, so daß vereinfachte Verfahren gesucht wurden, um ihn einigermaßen realistisch zu simulieren. Eines dieser Verfahren ist der sogenannte α-Buffer [KG79]. Er akkumuliert die Transparenzwerte der eingetragenen Objekte so, daß für wenige hintereinanderliegende Objekte der gewünschte Eindruck entsteht. α ist ein für jeden Punkt eines Objektes gegebener Wert, der 1 ist, wenn ein Objekt durchsichtig ist und 0, wenn es nicht durchsichtig ist.

Soll aufgrund des z-Vergleichs ein neuer Farbwert rgb_{neu} vor einen bereits im z-Buffer stehenden Farbwert rgb_{alt} geschrieben werden, dann berechnet sich der akkumulierte Farbwert zu

$$rgb = rgb_{neu} + \alpha_{neu} \cdot rgb_{alt},$$

der neue z-Wert wird eingetragen und der α-Wert wird modifiziert.

$$\alpha = (1 - \alpha_{neu}) + \alpha_{neu} \cdot (1 - \alpha_{alt})$$

Wird beispielsweise eine opake (undurchsichtige) Fläche A mit der Farbe C_A und der Transparenz $\alpha_A = 0$ in den Buffer geschrieben, so ist die neue Farbe:

$$C = C_A + \alpha_A C_{alt} = C_A$$

und der neue α-Wert:

$$\alpha = \alpha_A(1 - \alpha_{alt}) + (1 - \alpha_A) = 1.$$

Die Farbe wird also korrekt eingetragen. Für mehrere hintereinanderliegende transparente Flächen ist diese Methode allerdings nicht unbedingt geeignet, so daß für die Generierung realistischer Bilder doch besser auf die Strahlverfolgung zurückgegriffen werden sollte.

14. Beispiele zu den Darstellungsmethoden

Das Programm PRIM kann GED-Dateien mit dem Radiosity-Verfahren oder mit einer direkten Beleuchtung durch Lichtquellen berechnen und darstellen. Für beide Darstellungsarten verwendet es den z-Buffer-Algorithmus, um das Verdeckungsproblem zu lösen und damit die Darstellung zügig zu berechnen. Daher können Alternativen zum z-Buffer mit diesem Programm nicht ausprobiert werden. Allerdings können die verschiedenen Schattierungsverfahren und Texturierungen programmiert werden. Sie werden in diesem Kapitel eingeführt.

14.1 Schattierungsverfahren

In GED-Dateien sind die Angaben zur Schattierung ebenso wie für die Texturierung Objekteigenschaften. Sie können von Objekten auf abgeleitete Objekte „vererbt" werden. Dies steht, zumindest für die Schattierung, im Gegensatz zu vielen anderen Systemen, die diese Angabe nur für die ganze Szene festlegen.

Mit den Befehlen

```
SCHATTIERUNG FLACH;
SCHATTIERUNG GOURAUD;
SCHATTIERUNG PHONG;
```

wird die Schattierung eingestellt. Fehlt die Angabe, so wird als Schattierungsart PHONG verwendet. Wir wollen dazu diesmal einen TORUS als Beispiel nehmen. Mit ihm werden wir auch kurz auf das Problem der Facettierung und des Mach-Band-Effekts eingehen. Die Beispielszene lautet:

```
GED_VERSION 1;

OBJEKT MeinTorus;
  TORUS 0.4;
ENDOBJEKT;

GERICHTETELQ 1,1,1 1,1,-1;
SZENE Beispiel7;
  OD 0,1,0; ZEICHNE MeinTorus;
END.
```

Lassen Sie dieses Beispiel berechnen, so erscheint auf den Bildschirm ein schattierter, grüner Torus. Um die anderen Schattierungsarten auszuprobieren, wird

die Szene um zwei weitere Tori erweitert. Der eine erhält die Schattierung FLACH, der zweite GOURAUD.

```
GED_VERSION 1;

OBJEKT MeinTorus;
  TORUS 0.4;
ENDOBJEKT;

GERICHTETELQ 1,1,1 1,1,-1;
SZENE Beispiel8;
          OD 0,1,0; SCHATTIERUNG GOURAUD; ZEICHNE MeinTorus;
  VERSCHIEBE -2,-2,0; SCHATTIERUNG FLACH; ZEICHNE MeinTorus;
  VERSCHIEBE   4,4,0; SCHATTIERUNG PHONG; ZEICHNE MeinTorus;
END.
```

Je nach Grafikkarte, auf der Sie das Beispiel nun laufen lassen, sehen Sie mehr oder weniger deutlich folgendes: Der linke, vordere Torus erscheint zwar schattiert, die einzelnen Facetten sind aber deutlich erkennbar. Der mittlere ist zwar stetig im Farbverlauf, an Stellen mit starker Steigung sind aber die sogenannten Mach-Band-Effekte beobachtbar. Hier treten die Facetten teilweise wieder hervor. Der hintere Torus dagegen ist homogen in der Schattierung.

Für fast alle Szenen kann generell behauptet werden, daß Phong-schattierte Bilder besser als Gouraud-schattierte und diese wiederum besser als flach schattierte Bilder aussehen. Dies können Sie auch an Abb. II des farbigen Bildteils beobachten.

Der Zusammenhang zwischen Facettierung und Schattierung kann grob so formuliert werden:

> *Je feiner die Facettierung, desto „schlechter" kann die Schattierung gewählt werden.*

Ist die Facettierung beispielsweise so fein, daß ein Dreieck oder Viereck auf dem Bildschirm nur noch einem Pixel entspricht, dann kann dieses Polygon konstant schattiert werden [BFGS86]. Geben Sie beispielsweise für eine Standard-VGA-Karte die Facettierung des Torus mit

```
TORUS 0.4 FACET 40,20;
```

an, so sind kaum noch Unterschiede in der Darstellung der hinteren beiden Tori zu erkennen.

Mit zunehmender Feinheit der Facettierung gleicht sich auch die Erscheinungsweise des vorderen Torus derjenigen der hinteren beiden an.

14.2 Texturierung

In PRIM sind die drei Texturierungsarten Farbtextur-Mapping, Bumpmapping und Volumentexturen implementiert. Mit den Schlüsselworten

```
MAP
BUMPMAP
SOLIDMAP
NOMAP
```

werden sie, wie andere Objekteigenschaften, ein- bzw. ausgeschaltet. Für jedes Objekt gilt immer genau die letzte definierte Texturart. Voreingestellt ist keine Textur, also der Wert NOMAP.

Das Beispiel, anhand dessen die Texturierung diskutiert werden soll, ist die folgende GED-Datei:

```
GED_VERSION 1;

OBJEKT MeineEbene;
  PUNKTE
    0,0,0 1,1;
    0,1,0 0,1;
    0,1,1 0,0;
    0,0,1 1,0;
  FLAECHEN
    0,1,2,3;
ENDOBJEKT;

GERICHTETELQ 1,1,1 1,1,-1;
SZENE Beispiel9;
  DREHE x,-60;
  DREHE z,-90;
  VERSCHIEBE 0,-2.0,-2.0;
  SKALIERE 4.0;
  // BUMPMAP c:\prim\textures\bump.bmp;
  MAP c:\prim\textures\bear.bmp;
  // SOLIDMAP HOLZ;
  // SOLIDMAP MARMOR;
  ZEICHNE MeineEbene;
END.
```

Als Testobjekt wird hier das Objekt MeineEbene verwendet, das nicht nur aus vordefinierten Körpern, sondern erstmals aus selbst definierten Punkten und Flächen besteht. Es handelt sich hier um ein Viereck mit Texturkoordinatenangaben. Hinter dem Schlüsselwort PUNKTE stehen die Koordinatendefinitionen, hinter dem Schlüsselwort FLAECHEN die einzelnen Flächen. Beachten Sie, daß Punktindizes ab der Nummer 0 gezählt werden.

Die Ebene wird in der Szenendefinition geeignet rotiert, skaliert und verschoben, um sie vom Standardblickpunkt aus gut beobachten zu können. Mit den teilweise einkommentierten Zeilen werden Texturen für diese Fläche definiert. Als Volumentexturen sind nur die beiden prozedural erzeugten Texturen HOLZ und MARMOR vorgesehen.

Hinter den Schlüsselwortangaben MAP und BUMPMAP muß der *vollständige* Dateiname einer Windows-BMP-Datei stehen. MAP bewirkt ein Farbtexturmap-

ping; BUMPMAP die Interpretation des Dateiinhaltes als Bumpmap. Hierbei entspricht die Helligkeit des Bildes der Variationsgröße für den Normalenvektor. Experimentieren Sie mit anderen Dateien, den verschiedenen Texturarten und den Texturkoordinaten der Punkte.

Bei der gegebenen Definition wird die Textur genau ein Mal auf die Fläche abgebildet. Soll sie mehrfach oder gar verzerrt erscheinen, müssen die Texturkoordinaten entsprechend gewählt werden. Mit der Definition:

```
...
PUNKTE
  0,0,0 1,2;
  0,1,0 0,2;
  0,1,1 0,0;
  0,0,1 1,0;
...
```

erscheint die Textur in y-Richtung zwei Mal auf der Fläche.

Texturkoordinaten können, wie die Koordinaten auch, transformiert werden. Dies geschieht, indem zwischen einer Transformationsangabe und deren Werten [MIT] TEXTUR eingefügt wird. So bedeuten beispielsweise

```
SKALIERE TEXTUR 2;
SKALIERE MIT TEXTUR 2;
```

daß nur die Texturkoordinaten bzw. sowohl die Koordinaten als auch die Texturkoordinaten mit dem Faktor 2 skaliert werden.

Auf ein Problem sei an dieser Stelle noch hingewiesen: Da die Texturkoordinaten, genau wie die Farbwerte beim Gouraud-schattieren, linear interpoliert werden, können bei stark verzerrten Flächen auch die Texturen verzerrt erscheinen. Dem kann mit einer zusätzlichen Unterteilung der Flächen mit FACET bei der Definition oder während des Programmlaufs entgegengewirkt werden. Die Definition

```
...
FLAECHEN
  0,1,2,3 FACET 16;
...
```

im oben angeführten Beispiel bewirkt eine Unterteilung der Fläche in 16×16 Unterflächen. Die Texturkoordinaten werden bei der Unterteilung automatisch angepaßt..

Viele weitere Beispiele für die Verwendung von Texturen finden Sie in den GED-Dateien der Diskette und in den Abbildungen des farbigen Bildteils.

15. Implementierung der Darstellungsmethoden

Nach den eher theoretischen Betrachtungen zum Thema Darstellungsmethoden befaßt sich dieses Kapitel mit den Darstellungsmethoden „zum Anfassen", mit der Implementierung.

Für Programmierer durchaus üblich ist der Versuch, sich das Leben so einfach wie möglich zu gestalten. Dies läuft im allgemeinen darauf hinaus, Gemeinsamkeiten der verschiedenen Aufgaben, die zu realisieren sind, zu suchen und diese Gemeinsamkeiten in einem Algorithmus zusammenzufassen. Mit diesem Thema, bezogen auf die Darstellungsmethoden, beschäftigen wir uns im nächsten Abschnitt. Anschließend werden Interpolationen und die Scankonvertierung diskutiert.

15.1 Gemeinsamkeiten der Darstellungskonzepte

Sei es nun Flat-, Gouraud-, Phong oder texturiertes Phong-Shading: Bei all diesen Verfahren ist es nötig, eine bestimmte Anzahl von Parametern beim Rasterisieren der Polygone zu interpolieren. Am Pixel dienen diese Parameter dann als Eingangswerte für mehr oder weniger komplexe Berechnungen.

Beim Flat-Shading wird lediglich der z-Wert interpoliert. Für ein Pixel ist dann die Funktion

Algorithmus 15.1 (Flat-Shading)

WENN $Z_i < z\text{-}Buffer[x, y]$, *DANN:*
  $z\text{-}Buffer[x, y] := Z_i$.
  $FrameBuffer[x, y] :=$ *Farbe des Polygons.*
SONST tue nichts.

zu berechnen, wobei Z_i dem interpolierten z-Wert entspricht.

Beim Gouraud-Shading werden schon vier Parameter interpoliert: die Tiefe z, der rote, der grüne und der blaue Farbanteil r, g und b. Für das Gouraud-Shading ist dann am Pixel die Funktion

Algorithmus 15.2 (Gouraud-Shading)

WENN $Z_i < z\text{-}Buffer[x,y]$, *DANN:*
  $z\text{-}Buffer[x,y] := Z_i$.
  Rot-Anteil $FrameBuffer[x,y] := R_i$.
  Grün-Anteil $FrameBuffer[x,y] := G_i$.
  Blau-Anteil $FrameBuffer[x,y] := B_i$.
SONST tue nichts.

zu berechnen, wobei Z_i, R_i, G_i und B_i den interpolierten Werten entsprechen.

Beim Phong-Shading ist die Pixelfunktion nicht ganz so trivial wie die bisher vorgestellten. Genauso wie beim Gouraud-Shading sind vier Parameter zu interpolieren, dieses Mal z, n_x, n_y und n_z, wobei n_x, n_y und n_z die Komponenten des Normalenvektors sind. Wenn beim Phong-Shading die diffuse Farbe des Polygons ebenfalls interpoliert wird, sind es entsprechend sieben Parameter, nämlich zusätzlich die drei Farbanteile der diffusen Farbe. Für das Phong-Shading lautet der Algorithmus am Pixel wie folgt:

Algorithmus 15.3 (Phong-Shading)

WENN $Z_i < z\text{-}Buffer[x,y]$, *DANN:*
  $z\text{-}Buffer[x,y] := Z_i$.
  Bilde einen Vektor N *aus den Komponenten* $n_{x_i}, n_{y_i}, n_{z_i}$.
  Normiere N.
  Bilde den Punkt (x, y, Z_i) *vom Bildschirm ..*
  .. in's Weltkoordinatensystem ab.
  Wende das Beleuchtungsverfahren auf diesen Punkt unter ..
  .. Verwendung des Normalenvektors an.
  $FrameBuffer[x,y] :=$ *Ergebnis dieser Beleuchtungsberechnung.*
SONST tue nichts.

Der Normalenvektor N muß dann nicht in die Welt zurücktransformiert werden, wenn gleich seine Weltkoordinatenkomponenten interpoliert werden. Das ist zwar bei stark perspektivischen Verzerrungen nicht ganz korrekt, hat aber diverse andere Vorteile – nicht zuletzt erübrigt sich die Transformation, die ja pro Pixel einmal berechnet werden muß.

Somit ist eigentlich klar, was zu tun ist, wenn eine gemeinsame Routine für die verschiedenen Schattierungsverfahren verwendet werden soll: Man implementiert eine Rasterisierungsroutine, die nicht *einen*, sondern n Parameter interpoliert und ruft dann die Pixelfunktion indirekt über einen Zeiger auf, der zu Beginn je nach Schattierungsverfahren auf die entsprechende Pixelfunktion zeigt.

Ist so eine Routine erst einmal geschrieben, ist es leicht, neue Methoden zu implementieren. Soll zusätzlich zu der normalen Schattierung eine 2D-Textur auf das Polygon geklebt werden, sind nur zwei Stellen zu ändern:

1. Die Anzahl der zu interpolierenden Werte ist um 2 zu erhöhen. Diese beiden Werte entsprechen den Texturkoordinaten u und v. Die Anfangswerte sind entsprechend zu setzen.
2. Eine neue Pixelfunktion muß implementiert werden, die anhand der neuen und alten Parameter das neue Verfahren implementiert.

Soll z.B. das Gouraud-Shading mit einer 2D-Textur kombiniert werden, so ist die neue Pixelfunktion:

Algorithmus 15.4 (Gouraud-Shading mit Textur-Mapping)

WENN $Z_i < z\text{-}Buffer[x,y]$, *DANN:*
$z\text{-}Buffer[x,y] := Z_i$.
Lese Textur$[U_i, V_i]$.
Rot-Anteil FrameBuffer$[x,y] :=$ (R_i *gemischt mit .. .. dem Rotanteil der Textur).*
Grün-Anteil FrameBuffer$[x,y] :=$ (G_i *gemischt mit .. .. dem Grünanteil der Textur).*
Blau-Anteil FrameBuffer$[x,y] :=$ (B_i *gemischt mit .. .. dem Blauanteil der Textur).*
SONST tue nichts.

15.2 Interpolationen

Da wir uns in der Implementierung darauf beschränkt haben, maximal eine Textur pro Teilobjekt zuzulassen, ist die Maximalzahl der Werte, die interpoliert werden müssen, zehn. Diese Maximalzahl wird für Phong-schattierte Polygone (z, n_x, n_y, n_z), die mit einer Volumentextur versehen sind (t_x, t_y, t_z) und deren diffuse Reflektivität interpoliert werden soll (r, g, b), erreicht.

Da die Anzahl der zu interpolierende Werte also begrenzt ist, können die Werte in einem statischen Feld gespeichert werden, was die Implementierung deutlich vereinfacht. Je nach Schattierungs- und Texturart werden dann die ersten k der maximal zehn Werte interpoliert.

Es stellt sich die Frage, wie man die verschiedenen Werte in einem Feld speichern soll, wo sie doch i.a. verschiedene Datentypen haben. Hier bietet sich ein Ausweg an, der außer der Tatsache, daß alle zu interpolierenden Werte im gleichen Format darstellbar sind, noch den Vorteil hat, daß Interpolationen extrem schnell vom Prozessor ausgeführt werden: Die Abspeicherung der zu interpolierenden Werte als Festkommazahl.

Festkommazahlen sind eigentlich nichts anderes als gewöhnliche Ganzzahlen, die nur anders interpretiert werden. Hat bei einer Integerzahl das i-te Bit die Wertigkeit 2^i, so wird bei einer Festkommazahl die Wertigkeit als 2^{i-k} gedeutet, wobei k die Anzahl der Nachkommabits angibt. Wählt man für $k = 16$,

so können mit einer 32-Bit Integer-Zahl Werte von −32768 bis 32767 mit 16 Nachkommastellen dargestellt werden. Nun kann man mit Hilfe eines varianten Records auch noch besonders einfach auf solch eine Festkommazahl zugreifen:

```
FestKommaTYP = RECORD CASE BOOLEAN OF
                 TRUE : (Zahl     : LONGINT);
                 FALSE: (BruchTeil: WORD;
                         GanzTeil : INTEGER);
               END;
```

Dieser Typ ist in der Unit PRIMGLOB definiert und von zentraler Bedeutung bei der Interpolation.

Ein Beispiel: Die Zahlen 3.25 und 4.5 sollen als Festkommazahlen dargestellt und ihre Summe gebildet werden.

```
VAR
  a, b, c: FestKommaTYP;
BEGIN
  a.GanzTeil:=3; a.BruchTeil=$4000;  {$4000= 0.25*2^16}
  b.GanzTeil:=4; b.BruchTeil=$8000;  {$8000= 0.5 *2^16}
  c.Zahl:=a.Zahl+b.Zahl;
END.
```

c hat nach der Addition den Wert `c.GanzTeil=7` und `c.BruchTeil=$C000`, was dem erwarteten Wert entspricht ($C000=$0.75*2^{16}$).

Obwohl also reine Integerrechnung verwendet wird, ist es durchaus möglich, Nachkommaanteile zu berücksichtigen. Bei der eigentlichen Interpolation werden zwar die Nachkommaanteile berechnet, bei der Pixelfunktion reicht aber im allgemeinen die Verwendung der Vorkommaanteile.

Mit diesem Wissen ist es bereits möglich, die eigentliche Interpolation als Programm zu schreiben:

```
FOR i:=0 TO AnzahlZuInterpolierendeWerte-1 DO
  Werte[i].Zahl:=Werte[i].Zahl+Steigungen[i].Zahl;
```

oder kürzer:

```
FOR i:=0 TO AnzahlZuInterpolierendeWerte-1 DO
  INC(Werte[i].Zahl,Steigungen[i].Zahl);
```

Nicht nur, daß die Interpolation von Festkommawerten besonders einfach ist, auch die Multiplikation und die Division einer Festkommazahl mit einer *ganzen Zahl* ist nichts anderes als eine normale Integermultiplikation bzw. -division.

Will man in obigem Beispiel die Zahl b durch 5 teilen, so kann dies mit der Anweisung `b.Zahl:=b.Zahl DIV 5;` geschehen. Hatte b vor der Division den Wert 4.5, so ergibt sich für b nach der Division für `b.Zahl` der Wert $E666 bzw. für `b.GanzTeil` der Wert 0 und für `b.BruchTeil` der Wert $E666, was, mit 2^{-16} multipliziert, 0.89999389... ergibt. Dieser Wert entspricht ziemlich genau dem erwarteten Wert 0.9. Der maximale Fehler ist 2^{-16}.

Von dieser Tatsache wird insbesondere bei der Schattierung von Dreiecken, die Gegenstand des nächsten Abschnittes ist, Gebrauch gemacht.

15.3 Rasterisierung und Schattierung von Dreiecken

Bereits im Abschnitt 13.2 wurde kurz auf die Scankonvertierung von Polygonen eingegangen. Planare Polygone, die keine Dreiecke sind, können immer in solche zerlegt werden. In der Implementierung werden Dreiecke verwendet. Sie haben den Vorteil, daß sie

1. immer konvex sind und
2. die lineare Interpolation von Werten, die an den Eckpunkten des Dreiecks gegeben sind, für ein bestimmtes Pixel im Dreieck eindeutig ist.

Da Dreiecke immer konvex sind, existiert pro Scanlinie immer nur genau ein Teilstück, ein sogenannter Span, der von der linken bis zur rechten Kante des Dreiecks reicht. Das Dreieck wird mit Hilfe dieser Spans gefüllt, indem vom oberen bis zum unteren Punkt des Dreiecks Spans entsprechender Länge gezeichnet werden.

Zur Bestimmung von Werten im Inneren des Dreiecks, die aus den zugehörigen Werten an den Eckpunkten gewonnen werden (z.B. Farbwerte, Texturkoordinaten) kann ausgenutzt werden, daß eine Ebene durch drei Punkte immer eindeutig definiert ist. Für einen vierten Punkt – den Punkt im Inneren des Dreiecks – ist es dann möglich, den zugehörigen Wert zu bestimmen, indem der Wert der Ebene an der entsprechenden Stelle ausgewertet wird.

Letztlich läßt sich das Darstellen von Dreiecken in folgende Aufgaben gliedern:

1. Bestimmen der Spans, die das Dreieck definieren. Für jeden Span muß also seine y-Koordinate, sowie sein x-Anfangs- und x-Endwert bestimmt werden.
2. Bestimmen der interpolierten Werte für die im Punkt 1. erhaltenen Spans.

Diese beiden Aufgaben sind Gegenstand der folgenden beiden Abschnitte.

15.3.1 Rasterisierung von Dreiecken

Unter den Computergrafikern ist das Rasterisieren von Dreiecken „ein alter Hut". Die zu verwendenden Algorithmen gehören ebenso zu den Standardverfahren der Computergrafik wie das Zeichnen von Punkten und Linien. Trotzdem versuchen einige Firmen mit ziemlicher Penetranz, diese Verfahren als „streng geheimes innerbetriebliches Know-How" einzustufen.

Das dem nicht so ist, läßt sich allein aus der Vielzahl der zu diesem Thema veröffentlichten Literatur ersehen [FvDFH90,HP89,LKV90,Kir92,HA90]. Vor allem der in [LKV90] beschriebene Algorithmus gab den Anstoß für den hier implementierten Algorithmus. Das Besondere an ihm ist, daß nicht nur die ganzzahligen Anteile der Eckpunktkoordinaten, sondern auch ihre Nachkommaanteile berücksichtigt werden, um zu bestimmen, welches Pixel im Inneren eines Dreiecks liegt.

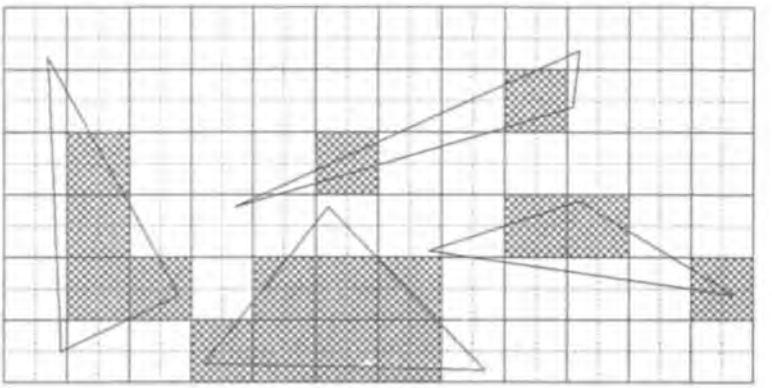


Abb. 15.1 Dreiecke, bei denen nur solche Pixel gesetzt werden, deren Mittelpunkte innerhalb des Dreiecks liegen

Ein Pixel ist nicht unendlich klein. In Bildschirmkoordinaten hat es die Breite und Länge 1. Verwendet man kein Antialiasing (siehe Abschnitt 7.8), so kann dieses Pixel nur gesetzt oder nicht gesetzt werden. Im implementierten Algorithmus werden genau solche Pixel gesetzt, deren *Pixelmittelpunkte* im Inneren des Dreiecks liegen. Die Koordinaten dieser Pixelmittelpunkte werden als ganzzahlig definiert. Das heißt z.B. für das Pixel in der oberen linken Ecke des Bildschirms, daß es die Mittelpunktkoordinaten $(0, 0)$ hat. Da es außerdem die Länge und Breite 1 hat, folgt, daß der linke Rand des Bildschirms die x-Koordinate -0.5 und der obere Rand die y-Koordinate -0.5 hat.

Diese Überlegungen sind zur Bestimmung der Transformationsmatrizen wichtig, die Punkte auf Bildschirmkoordinaten abbilden. Das Bildschirmfenster ist für solche Abbildungen entsprechend zu setzen.

In Abb. 15.1 sind einige Dreiecke dargestellt, bei denen die nach dieser Definition zu setzenden Pixel gekennzeichnet sind. Diese Definition, nach der Dreiecke „subpixelpositioniert" werden, hat folgende Vorteile:

1. In einem aus Dreiecken zusammengesetzten Gebilde werden die zu setzenden Pixel eindeutig genau einem Dreieck zugeordnet. Sie werden also nur ein Mal gesetzt, so daß es weder zu Überlappungen (overlaps) noch zu Löchern (drop outs) kommt.
   Sich überlappende Dreiecke würden speziell dann, wenn sie transparent sind, zu Artefakten führen. Die Pixel, die von mehr als einem Dreieck gesetzt würden, erschienen nämlich mit der falschen Intensität. Hält man zwei getönte Glasscheiben nebeneinander, wobei sich die Ränder der Scheiben ein wenig überlappen, so erscheinen die Überlappungen dunkler, da dort das Licht nicht nur von einer, sondern von beiden Scheiben gedämpft wird.
2. Die Fläche, die die Pixel bedecken, ist, genügend große Dreiecke vorausgesetzt, eine gute Näherung für den tatsächlichen Flächeninhalt des

Dreiecks. Dies ist besonders bei der Formfaktorberechnung mit Hilfe des Hemicubes oder der Singleplane (siehe Teil 3) wichtig.

Grob formuliert hat der Algorithmus zum Rasterisieren eines Dreiecks folgende Form:

Algorithmus 15.5 (Rasterisieren eines Dreiecks)

Sortiere die übergebenen Eckpunkte nach ihrer y-Koordinate.
FÜR alle y des Teildreiecks vom oberen zum mittleren Punkt:
Zeichne einen Span von der langen zur kurzen Kante.
FÜR alle y des Teildreiecks vom mittleren zum unteren Punkt:
Zeichne einen Span von der langen zur kurzen Kante.

Der erste Span des oberen Teildreiecks liegt in Höhe der aufgerundeten y-Koordinate des obersten Eckpunktes, der letzte bei der abgerundeten y-Koordinate des mittleren Eckpunktes.

Der erste Span des unteren Teildreiecks liegt in Höhe der aufgerundeten y-Koordinate des mittersten Eckpunktes, der letzte bei der abgerundeten y-Koordinate des unteren Eckpunktes. Unter Umständen existieren für das obere und/oder das untere Dreieck keine Spans.

Anfangs- und Endpunkt der Spans werden aufgrund der Kanten bestimmt. Der Pixelmittelpunkt des linken Span-Anfangspunktes muß dabei gerade rechts der linken Kante und der rechte gerade links der rechten Kante liegen. Diese Anfangs- und Endpunkte werden mit Hilfe eines modifizierten Bresenham-Algorithmus bestimmt.

15.3.2 Bestimmung der linken Span-Anfangspunkte

Soll die linke x-Koordinate eines in der Höhe y liegenden Spans bestimmt werden, so kann dies mit folgendem „Trick" geschehen:

Man erhöht in Gedanken die Auflösung des Bildschirms, so daß jedes Pixel in $2^{16} \cdot 2^{16}$ Unterpixel aufgeteilt wird. Die unteren 16 Bit einer Ganzzahlkoordinate geben dann die Position innerhalb eines Pixels, die restlichen die Koordinate des ursprünglichen Bildschirms an.

Man berechnet sodann die Steigung dx/dy der linken Kante des „vergrößerten" Dreiecks, die identisch mit der Steigung des ursprünglichen Dreiecks ist, beginnt in Höhe des obersten Spans und addiert sukzessive $2^{16} \cdot dx/dy$. Auf diese Weise erhält man die Pixel- und Subpixelkoordinaten der linken Kante in Höhe der Spans. Ist der Subpixelkoordinatenanteil größer als 2^{15}, entspricht er also 0.5, so liegt der Anfangswert des entsprechenden Spans bei der errechneten Pixelkoordinate +1, sonst bei der errechneten Pixelkoordinate.

Als Algorithmus formuliert:

Algorithmus 15.6 (Bestimme x-Anfangswerte eines Spans)

Bilde die „vergrößerten" Koordinaten von Anfangs- und Endpunkt ..
.. der linken Kante: ..
.. $x_1 = p1_x \cdot 2^{16}, y_1 = p1_y \cdot 2^{16}, x_2 = p2_x \cdot 2^{16}, y_2 = p2_y \cdot 2^{16}$.
Bestimme die Schrittweite zu $(p2_x - p1_x)/(p2_y - p1_y) \cdot 2^{16}$.
Berechne die x- und y-Koordinate des obersten Spans zu ..
.. $y_o = p1_y$ *aufgerundet und* $x_o = p1_x + dx/dy \cdot (p1_y - y)$.
Berechne die y-Koordinate des untersten Spans zu $y_u = p2_y$ *abgerundet.*
Bestimme die „vergrößerte" integer x-Koordinate des obersten Spans ..
.. aus der unvergrößerten durch $x'_o = \text{round}(x_o \cdot 2^{16})$.
FÜR alle ganzzahligen y von y_o *bis* y_u*:*
  $x_s =$ *Bits von* x'_o *ab Bit 16.*
  WENN die unteren 16 Bit von $x'_o > 2^{15}$*, DANN:*
    $x_s \leftarrow x_s + 1$.
  Linkes Pixel des Spans ist bei (x_s, y).
  $x'_o \leftarrow x'_o +$ *Schrittweite.*
ENDE FÜR alle.

Nun kann man die oberen Bits von x'_o und *Schrittweite* von den unteren 16 Bit trennen und diese separat berechnen. Die unteren 16 Bit von x'_o dienen dabei im Prinzip als Fehlerterm.
Mit:

x_{oh} :	Integerzahl, entspricht x'_o ab Bit 16
Fehler :	Integerzahl, entspricht den unteren 16 Bit von x'_o
$Delta_h$:	Integerzahl, entspricht der Schrittweite ab Bit 16
Delta :	Integerzahl, entspricht den unteren 16 Bit der Schrittweite

ergibt sich ein Algorithmus, der getrennt nach Pixel- und Subpixelanteilen arbeitet.

Algorithmus 15.7 (Subpixelpositionierte Rasterisierung (1))

Berechne Schrittweite, y_o, x_o, y_u, x'_o *wie im vorigen Algorithmus.*
$x_{oh} \leftarrow x'_o$ *16 nach rechts verschoben.*
Fehler $\leftarrow x'_o$ *bitweise UND mit $FFFF.*
$Delta_h \leftarrow$ *Schrittweite 16 nach rechts verschoben.*
Delta $\leftarrow$ *Schrittweite bitweise UND mit $FFFF.*
FÜR alle ganzzahligen y von y_o *bis* y_u*:*
  $x_s \leftarrow x_{oh}$.
  WENN Fehler $> 2^{15}$*, DANN:*
    $x_s \leftarrow x_s + 1$.
  Linkes Pixel des Spans ist bei (x_s, y).
  $x_{oh} \leftarrow x_{oh} + Delta_h$.
  Fehler $\leftarrow$ *Fehler + Delta.*
  WENN Fehler $> 2^{16}$*, DANN:*
    Fehler $\leftarrow$ *Fehler* -2^{16}.
    $x_{oh} \leftarrow x_{oh} + 1$.
  ENDE WENN.
ENDE FÜR alle.

Zu x_{oh} wird also pro Durchlauf $Delta_h$ addiert. Ist der Fehler nach der Addition von *Delta* größer als 2^{16}, wird zusätzlich 1 addiert.

Noch zwei sinnvolle Umformungen führen zu dem letztlich implementierten Algorithmus:

1. Verschiebt man die Kante vor Beginn der Berechnung um 0.5 nach rechts, so kann der erste Vergleich der Schleife eingespart werden. Es ergeben sich trotzdem die gleichen Werte für x_s, wenn man vor der Schleife zu x_{oh} 1 addiert.
2. Führt man zusätzlich zu *Delta* und $Delta_h$ die Variablen $Delta_+$ und $Delta_{h+}$ ein, die sich zu $Delta_+ = Delta - 2^{16}$ und $Delta_{h+} = Delta_h + 1$ ergibt, so kann die Subtraktion von 2^{16} von *Fehler* und die Addition von 1 zu x_{oh} „versteckt“ werden.

Letztlich erhält man:

Algorithmus 15.8 (Subpixelpositionierte Rasterisierung (2))

Bestimme die Schrittweite zu $(p2_x - p1_x)/(p2_y - p1_y) \cdot 2^{16}$.
Berechne die x- und y-Koordinate des obersten Spans zu $y_o = p1_y$ ..
.. *aufgerundet und* $x_o = p1_x + dx/dy \cdot (p1_y - y)$.
Berechne die y-Koordinate des unteren Spans zu $y_u = p2_y$ *abgerundet.*
Bestimme die „vergrößerte" integer x-Koordinate des obersten Spans ..
.. *aus der unvergrößerten durch* $x'_o = \text{round}(x_o \cdot 2^{16})$.
Addiere 2^{15} *zu* x'_o.
$x_{oh} \leftarrow$ (x'_o *16 nach rechts verschoben) + 1.*
$Delta_h \leftarrow$ *Schrittweite 16 nach rechts verschoben.*
$Delta_{h+} \leftarrow Delta_h + 1$.
$Delta \leftarrow$ *Schrittweise bitweise UND mit $FFFF.*
$Delta_+ \leftarrow Delta - 2^{16}$.
$Fehler \leftarrow$ (x'_o *bitweise UND mit $FFFF) + Delta.*
FÜR alle ganzzahligen y von y_o *bis* y_u*:*
  Linkes Pixel des Spans ist bei (x_{oh}, y).
  WENN $Fehler > 2^{16}$, *DANN:*
    $Fehler \leftarrow Fehler + Delta_+$.
    $x_{oh} \leftarrow x_{oh} + Delta_{h+}$.
  SONST:
    $Fehler \leftarrow Fehler + Delta$.
    $x_{oh} \leftarrow x_{oh} + Delta_h$.
  ENDE WENN, SONST.
ENDE FÜR alle.

Hat ein bestimmter Span also die Anfangskoordinaten (x_{oh}, y), so hat der nächste Span die Anfangskoordinaten $(x_{oh}+Delta_{h+}, y+1)$ oder $(x_{oh}+Delta_h, y+1)$, je nach dem Wert von *Fehler*. Von dieser Tatsache wird im Abschnitt über das Schattieren von Dreiecken Gebrauch gemacht.

Mit dem gerade hergeleiteten Algorithmus sind also die linken Koordinaten der Spans bekannt. Für die rechten Koordinaten ist der Algorithmus bis auf eine Ausnahme identisch: Hier werden nicht die Pixel gerade rechts der Kante, sondern die gerade links der Kante gesucht. Dies kann einfach dadurch geschehen, daß vor Beginn der Schleife von x_{oh} 1 subtrahiert wird. Liegt (x, y) gerade rechts von einer Kante, so liegt $(x-1, y)$ gerade links von ihr.

Aus der Initialisierung für x_{oh} wird

$x_{oh} \leftarrow$ x'_o *16 nach rechts verschoben,*

sonst ändert sich nichts.

Mit Hilfe der Routinen für die linke und rechte Kante, die parallel laufen, kann nun sowohl das obere als auch das untere Teildreieck gefüllt werden. Beim Wechsel vom oberen auf das untere Teildreieck muß die Kanteninterpolation für die untere kurze Kante des Dreiecks initialisiert werden, die Interpolation der langen Kante kann unverändert weiter ausgeführt werden.

Zu guter letzt bleibt noch die Frage, ob die lange (besser: die in Richtung y längste Kante) die linke oder die rechte Dreieckskante ist. Es muß schließlich für die linke Kante eine andere Initialisierung als für die rechte Kante vorgenommen werden. Die Frage kann entschieden werden, indem die Lage des mittleren Eckpunkts $p2$ relativ zur langen Kante bestimmt wird. Ist er rechts von ihr, so ist die lange Kante die linke, sonst die rechte. Liegt er auf der langen Kante, so ist das Dreieck entartet. Es hat den Flächeninhalt 0.

Um die horizontale Lage von $p2$ zur langen Kante zu bestimmen, berechnet man die Differenz der x-Koordinaten von $p2$ und dem Punkt der langen Kante in Höhe $p2_y$. Je nach Vorzeichen ergibt sich die gewünschte Antwort.

Die x-Koordinate x_L der langen Kante in Höhe $p2_y$ ist:

$$x_L = p1_x + (p2_y - p1_y) \cdot \frac{p3_x - p1_x}{p3_y - p1_y}$$

Mißt man von x_L aus nach rechts in positiver Richtung, so ergibt sich als Abstand

$$D = p2_x - x_L$$

Ist D positiv, so liegt $p2$ rechts der langen Kante, sonst links.

Mit den Schreibweise $x_{ij} := pi_x - pj_x$ und $y_{ij} := pi_y - pj_y$ kann D kürzer als

$$D = x_{21} - y_{21} \cdot \frac{x_{31}}{y_{31}}$$

geschrieben werden.

Multipliziert man beide Seiten dieser Gleichung mit y_{31}, so ändert sich zwar der Wert des Ausdrucks, nicht aber sein Vorzeichen, da y_{31} nach dem Sortieren der Eckpunkte positiv ist. Es kann also auch dieser Wert zur „Entscheidungsfindung“ herangezogen werden. Aus Gründen, die im nächsten Abschnitt deutlich werden, haben wir diesen Wert schlicht *Nenner* getauft:

$$Nenner = x_{21} \cdot y_{31} - x_{31} \cdot y_{21}$$

Das folgende Listing zeigt den Rasterisierungsteil unserer Implementierung. Er wird in ihr zwei Mal verwendet: einmal in der Unit `PRIMRAD` zum Rendern der Elemente auf die Singleplane (siehe Abschnitt 18.4.7), und einmal im Modul `PRIMSHD` zum Darstellen von Dreiecken auf dem Bildschirm. Die beiden Routinen unterscheiden sich von der hier abgedruckten dadurch, daß in `PRIMRAD` zusätzlich z und in `PRIMSHD` zusätzlich z, Farbe, Normale und Texturkoordinaten interpoliert werden (siehe Abschnitt 13.2).

```
PROCEDURE RasterisiereDreieck(x1, y1, x2, y2, x3, y3: SINGLE);
CONST Single2Fix = 65536.0;
      Fix2Single = 1.0/65536.0;

VAR x31,y31,x21,y21,x32,y32,h,Nenner,Steigung       : SINGLE;
    i,xAnfang,xEnde,x,y,yNeu, yUnten                : INTEGER;
    xOben,yOben, xMitte,yMitte                      : FestKommaTYP;
    FehlerLangeKante,       FehlerKurzeKante        : LONGINT;
    Delta1FehlerLangeKante, Delta1FehlerKurzeKante  : LONGINT;
    Delta2FehlerLangeKante, Delta2FehlerKurzeKante  : LONGINT;
    Schritt1LangeKante,     Schritt1KurzeKante      : INTEGER;
    Schritt2LangeKante,     Schritt2KurzeKante      : INTEGER;
    SteigungLangeKante,SteigungKurzeKante,xBeiyOben : FestkommaTYP;
    vertikaleVerschiebung                           : SINGLE;

  PROCEDURE Span(xAnfang, xEnde, y: INTEGER);
  VAR x:INTEGER;
  BEGIN
    IF Nenner>0 THEN FOR x:=xAnfang TO xEnde DO SetzePixel(x,y)
    ELSE FOR x:=xAnfang DOWNTO xEnde DO SetzePixel(x,y);
  END;

  PROCEDURE TeilDreieck(yVon, yBis: INTEGER);
  VAR y: INTEGER;
  BEGIN
    FOR y:=yVon TO yBis DO BEGIN
      Span(xAnfang,xEnde,y);
      IF FehlerLangeKante<$10000 THEN BEGIN
        INC(xAnfang,Schritt1LangeKante);
        INC(FehlerLangeKante,Delta1FehlerLangeKante);
      END ELSE BEGIN
        INC(xAnfang,Schritt2LangeKante);
        INC(FehlerLangeKante,Delta2FehlerLangeKante);
      END;
      IF FehlerKurzeKante<$10000 THEN BEGIN
        INC(xEnde,Schritt1KurzeKante);
        INC(FehlerKurzeKante,Delta1FehlerKurzeKante);
      END ELSE BEGIN
        INC(xEnde,Schritt2KurzeKante);
        INC(FehlerKurzeKante,Delta2FehlerKurzeKante);
      END;
    END;
  END;

BEGIN
  IF y1>y2 THEN BEGIN h:=y1; y1:=y2; y2:=h; h:=x1; x1:=x2; x2:=h; END;
  IF y1>y3 THEN BEGIN h:=y1; y1:=y3; y3:=h; h:=x1; x1:=x3; x3:=h; END;
  IF y2>y3 THEN BEGIN h:=y2; y2:=y3; y3:=h; h:=x2; x2:=x3; x3:=h; END;
  y21:=y2-y1; y31:=y3-y1; y32:=y3-y2;
  x21:=x2-x1; x31:=x3-x1; x32:=x3-x2;
  xOben.Zahl  := trunc(x1*$10000); yOben.Zahl  := trunc(y1*$10000);
  xMitte.Zahl := trunc(x2*$10000); yMitte.Zahl := trunc(y2*$10000);
  yUnten      := trunc(y3);

  IF y31<0.00001 THEN exit;
```

```
  Nenner := y31*x21-y21*x31; IF abs(Nenner)<0.00001 THEN exit;

  Steigung                  := x31/y31 * Single2Fix;
  SteigungLangeKante.Zahl   := trunc(Steigung);
  y := yOben.GanzTeil+1; yNeu := yMitte.GanzTeil+1;
  vertikaleVerschiebung     := 1-yOben.BruchTeil*Fix2Single;
  xBeiyOben.Zahl:=trunc(Steigung*vertikaleVerschiebung)+xOben.Zahl;

  FehlerLangeKante          := LONGINT(xBeiyOben.BruchTeil)+
                               SteigungLangeKante.BruchTeil;
  Delta1FehlerLangeKante    := SteigungLangeKante.BruchTeil;
  Delta2FehlerLangeKante    := Delta1FehlerLangeKante-$10000;
  Schritt1LangeKante        := SteigungLangeKante.GanzTeil;
  Schritt2LangeKante        := Schritt1LangeKante+1;
  xAnfang                   := xBeiyOben.GanzTeil;
  IF Nenner>0 THEN INC(xAnfang);

  IF yNeu>y THEN BEGIN
    Steigung                := x21/y21 * Single2Fix;
    SteigungKurzeKante.Zahl := trunc(Steigung);
    xBeiyOben.Zahl          := trunc(Steigung*vertikaleVerschiebung) +
                               xOben.Zahl;
    FehlerKurzeKante        := LONGINT(xBeiyOben.BruchTeil) +
                               SteigungKurzeKante.BruchTeil;
    Delta1FehlerKurzeKante  := SteigungKurzeKante.BruchTeil;
    Delta2FehlerKurzeKante  := Delta1FehlerKurzeKante-$10000;
    xEnde                   := xBeiyOben.GanzTeil;
    IF Nenner<0 THEN INC(xEnde);
    Schritt1KurzeKante      := SteigungKurzeKante.GanzTeil;
    Schritt2KurzeKante      := Schritt1KurzeKante + 1;
    TeilDreieck(y,yNeu-1);
    y:=yNeu;
  END;

  yNeu:=yUnten+1;
  IF yNeu>y THEN BEGIN
    Steigung                := x32/y32 * Single2Fix;
    SteigungKurzeKante.Zahl := trunc(Steigung);
    vertikaleVerschiebung   := 1 - yMitte.BruchTeil*Fix2Single;
    xBeiyOben.Zahl          := trunc(Steigung*vertikaleVerschiebung) +
                               xMitte.Zahl;
    FehlerKurzeKante        := LONGINT(xBeiyOben.BruchTeil) +
                               SteigungKurzeKante.BruchTeil;
    Delta1FehlerKurzeKante  := SteigungKurzeKante.BruchTeil;
    Delta2FehlerKurzeKante  := Delta1FehlerKurzeKante-$10000;
    xEnde                   := xBeiyOben.GanzTeil;
    IF Nenner<0 THEN INC(xEnde);
    Schritt1KurzeKante:=SteigungKurzeKante.GanzTeil;
    Schritt2KurzeKante:=Schritt1KurzeKante+1;
    TeilDreieck(y,yNeu-1);
  END;
END;
```

15.3.3 Bilineare Interpolation in einem Dreieck

Nachdem im letzten Abschnitt geklärt wurde, welche Pixel zu einem Dreieck gehören, behandelt dieser Abschnitt die Frage, welchen Wert diese Pixel besitzen. Je nach Interpretation kann dieser Wert die Tiefe, Farbe, Normale o.ä. sein. Gemeinsam ist diesen Werten, daß sie lediglich an den Eckpunkten des Dreiecks gegeben sind. Für jedes Pixel im Inneren muß er aus den Eckpunkten durch Interpolation errechnet werden.

Betrachtet man den an den Eckpunkten übergebenen Wert (in unserem Beispiel die Farbe) als dritte Koordinaten-Komponente der Eckpunkte, so erhält man drei dreidimensionale Punkte. Diese Punkte spannen eine Ebene im „Raum" auf. Für jedes Pixel im Dreieck wird nun der Wert gewählt, den die Ebene bei den (x, y)-Koordinaten dieses Punktes hat.

Wie im vorigen Abschnitt erwähnt, ist ein Pixel nicht unendlich klein. Das heißt wiederum, daß dem Pixel nicht ein Punkt, sondern ein Flächenstück auf der Ebene entspricht. Welcher Wert dieses Flächenstückes für das Pixel gewählt wird, ist implementierungsabhängig. Wir haben den Wert gewählt, der dem Pixelmittelpunkt entspricht.

Die Zuordnung zwischen Pixelmittelpunkt und Ebene ist immer eindeutig. Würde man irgendeinen Wert des Flächenstücks als Interpolationsergebnis wählen, so kann dies zu erheblichen Fehlern führen. So kann es an den Ränder auftreten, daß Werte der Ebene gewählt werden, die nicht in der Umrandung des Dreiecks liegen. Aus der Interpolation wird dann eine Extrapolation. Dies kann wiederum zur Folge haben, daß Über- oder Unterläufe auftreten.

Ist z.B. ein Dreieck an einer Kante schwarz und am gegenüberliegenden Punkt weiß, so bedeutet eine Extrapolation über die lange Kante hinaus, daß Farben „schwärzer als schwarz" berechnet werden. Wählt man dagegen nur die Werte der Pixelmittelpunkte und setzt außerdem nur die Pixel, deren Pixelmittelpunkte im Inneren des Dreiecks liegen, so sind solche Fehler ausgeschlossen.

Da alle zu berechnenden Werte auf einer Ebene liegen, kann ihre Bestimmung relativ einfach erfolgen, indem man die Steigungen der Ebene bezogen auf x und y berechnet. Diese Steigungen sind nichts anderes als die partiellen Ableitungen der Ebene. Die Berechnung der partiellen Ableitungen wird im folgenden beispielhaft für einen Wert c durchgeführt.

Wenn bekannt ist, um wieviel sich c ändert, wenn sich x bzw. y um 1 ändern, ist es leicht möglich, die Pixel des Dreiecks mit einem c-Wert zu versehen. In Abb. 15.2 ist ein Dreieck sowie die beiden zusätzlichen Punkte P_4 und P_5 dargestellt. Kennt man für diese beiden Punkte die Farbe P_{4c} und P_{5c}, so ergibt sich die horizontale Steigung c_H und die vertikale Steigung c_V durch die Differenzenquotienten

$$c_H = \frac{P_{4c} - P_{3c}}{P_{4x} - P_{3x}} \qquad \text{und} \qquad c_V = \frac{P_{3c} - P_{5c}}{P_{3y} - P_{5y}}$$

Für P_4 und damit für c_H ergibt sich:

$$P_{4x} = P_{1x} + \frac{x_{21}}{y_{21}} y_{31}, \qquad P_{4y} = P_{3y}, \qquad P_{4c} = P_{1c} + \frac{c_{21}}{y_{21}} y_{31}$$

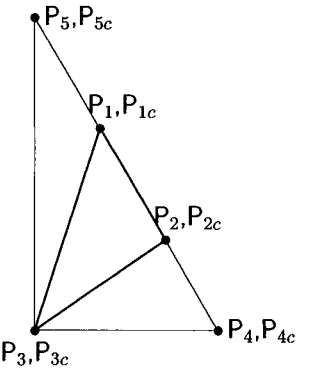


Abb. 15.2 Zur Bestimmung von c_H und c_V

$$
\begin{aligned}
c_H = \frac{dc}{dx} &= \frac{P_{4c} - P_{3c}}{P_{4x} - P_{3x}} \\
&= \frac{P_{1c} + \frac{c_{21}}{y_{21}} y_{31} - P_{3c}}{P_{1x} + \frac{x_{21}}{y_{21}} y_{31} - P_{3x}} \\
&= \frac{(P_{1c} - P_{3c}) y_{21} + c_{21} \cdot y_{31}}{(P_{1x} - P_{3x}) y_{21} + x_{21} \cdot y_{31}} \\
&= \frac{c_{21} \cdot y_{31} - c_{31} \cdot y_{21}}{x_{21} \cdot y_{31} - x_{31} \cdot y_{21}}
\end{aligned}
$$

Entsprechendes wird für P_5 und c_V angewendet.

$$
\begin{aligned}
P_{5x} &= P_{3x}, P_{5y} = P_{1y} + \frac{y_{21}}{x_{21}} x_{31}, P_{5c} = P_{1c} + \frac{c_{21}}{x_{21}} x_{31} \\
c_V = \frac{dc}{dy} &= \frac{P_{3c} - P_{5c}}{P_{3y} - P_{5y}} \\
&= \frac{P_{3c} - P_{1c} + \frac{c_{21}}{x_{21}} \cdot x_{31}}{P_{3y} - P_{1y} + \frac{y_{21}}{x_{21}} \cdot x_{31}} \\
&= \frac{c_{31} \cdot x_{21} - c_{21} \cdot x_{31}}{y_{31} \cdot x_{21} - x_{31} \cdot y_{21}}
\end{aligned}
$$

Verwendet man den im vorigen Abschnitt bereits eingeführten Wert *Nenner*, so lassen sich c_H und c_V auch durch

$$
\begin{aligned}
c_H &= \frac{c_{21} \cdot y_{31} - c_{31} \cdot y_{21}}{Nenner} \\
c_V &= \frac{c_{31} \cdot x_{21} - c_{21} \cdot x_{31}}{Nenner}
\end{aligned}
$$

ausdrücken.

Diese beiden Steigungen können nun gemeinsam mit dem Rasterisierungsalgorithmus verwendet werden, um die Werte pro Pixel zu berechnen. Hierfür

berechnet man den Wert des allerersten Pixels des ersten Spans und leitet hieraus den Wert aller weiteren Pixel ab. Dies bedeutet für die Spans, daß von Pixel zu Pixel jeweils c_H addiert werden muß, wenn der Span von links nach rechts gezeichnet wird. Entsprechend muß c_H subtrahiert werden, wenn der Span von links nach rechts geht.

Als Anfangswert der Spans wird das Ergebnis der Kanteninterpolation gewählt. Dieses erhält man, indem man von Kantenpixel zu Kantenpixel entweder $Delta_{h+} \cdot c_H + c_V$ oder $Delta_h \cdot c_H + c_V$ addiert, je nachdem, ob für den Kantenschritt $(Delta_{h+}, 1)$ oder $(Delta_h, 1)$ addiert wird. Aus Gründen der Einfachheit wird immer die lange Kante für die Kanteninterpolation der Werte verwendet.

Da die Werte an den Pixelmittelpunkten im Gegensatz zu den Koordinaten i.a. nicht ganzzahlig sind, wird für ihre Darstellung das Festkommaformat gewählt.

Zuletzt bleibt noch die Frage zu klären, welchen Wert das erste Pixel des ersten Spans hat. Hat der obere Eckpunkt des Dreiecks die Werte $(x1, y1, c1)$ und das erste Pixel die Koordinaten (x, y), so ergibt sich für dessen Wert

$$c = c1 + (x - x1) * c_H + (y - y1) * c_V.$$

15.4 Die Implementierung von Schattierung und Rasterisierung

Im Modul `PRIMSHD` sind die in diesem Kapitel beschriebenen Funktionen implementiert. Im Interfaceteil dieser Unit sind vier Prozeduren definiert, die allerdings nicht direkt, sondern nur indirekt durch andere Module, unter anderem von `PRIMREND` (siehe Abschnitt 9.3), angesprochen werden.

Die im Schnittstellendeklaration von `PRIMSHD` definierten Prozeduren sind:

```
PROCEDURE SetzeSchattierungsParameter(..)
PROCEDURE StelleFlaecheDar(..)
PROCEDURE ZeichneHintergrund(..)
PROCEDURE ZeichneDreieck(..)
```

`StelleFlaecheDar` ist die eigentliche Hauptroutine dieses Moduls. `SetzeSchattierungsParameter` setzt lediglich die Parameter, die bei der Schattierung aller Flächen konstant sind, so daß sie nicht für jede Fläche einzeln neu übergeben werden müssen. Dieses Verfahren spart Zeit beim Aufruf von `StelleFlaecheDar` und dient in der aufrufenden Prozedur `RendereSzene` aus `PRIMREND` der Übersichtlichkeit.

`ZeichneHintergrund` zeichnet den Hintergrund, was im einfachsten Fall bedeutet, daß der Bildschirm gelöscht wird. Mit dieser Routine sind auch weiche Farbverläufe und Hintergrundbilder möglich.

`ZeichneDreieck` wird über Umwege von der Unit `PRIMSHD` selbst aufgerufen. Flächen, die teilweise sichtbar sind, übergibt `StelleFlächeDar` der Unit

PRIMCLIP, die dann, nachdem die Fläche geclippt ist, ZeichneDreieck aufruft. Für Flächen, die komplett innerhalb des Sichtbereiches liegen, umgeht StelleFlaecheDar die Unit PRIMCLIP und ruft selbst ZeichneDreieck auf.

15.4.1 Die Prozedur SetzeSchattierungsParameter

Wegen der Kürze der Prozedur und ihrer Einfachheit steht hier statt langer Erläuterungen einfach das komplette Listing:

```
PROCEDURE SetzeSchattierungsParameter(VAR WC_DC, NPC_DC: Mat4TYP;
                    Lichtquellen: LichtquellenPtrTYP;
                    Auge: Vektor3DTYP;
                    FarbeVorberechnet: BOOLEAN;
                    DiffVorZuDiff, SpekGewichtung: SINGLE;
                    VorBerechnetesAmbiente: FarbVektorTYP);
BEGIN
  InvertiereMatrix(WC_DC, AktDC_WC);
  AktNPC_DC                := NPC_DC;
  AktLichtquellen          := LichtQuellen;
  AktAuge                  := Auge;
  AktDiffVorZuDiff         := DiffVorZuDiff;
  AktSpekGewichtung        := SpekGewichtung;
  AktFarbeVorberechnet     := FarbeVorberechnet;
  AktVorberechnetesAmbiente := VorberechnetesAmbiente;
END;
```

Bleibt die Frage, warum genau diese Variablen bei der Schattierung nötig sind.

AktDC_WC wird beim Phong-Shading für die Transformation der interpolierten Pixelkoordinaten in die Weltkoordinaten benötigt. AktNPC_DC leitet die Schattierungsroutine für zu clippende Polygone an die Unit PRIMCLIP weiter. Mit Hilfe dieser Matrix ist es dort möglich, beim Clippen neu erzeugte Punkte in Bildschirmkoordinaten zu transformieren.

AktLichtquellen und AktAuge kommen dann zur Geltung, wenn die Punkte des zu schattierenden Polygons noch beleuchtet werden müssen. Das ist einerseits beim Phong-Shading für jedes Pixel im Inneren der Fläche der Fall. Andererseits werden scharfe Punkte, also Punkte, die Eckpunkte von mehreren Polygonen sind und jeweils deren Normale annehmen, erst direkt vor dem Schattieren beleuchtet. Auch für diese scharfen Punkte sind AktLichtquellen und AktAuge notwendig. Die scharfen Punkte wurden im Zusammenhang mit der Unit PRIMREND in Abschnitt 9.3 näher erläutert.

Die letzten vier Werte, AktDiffVorZuDiff, AktSpekGewichtung, AktFarbeVorberechnet und AktVorberechnetesAmbiente dienen zum Mischen von vorberechneten Farben mit den beim Beleuchten eines Punktes entstandenen Farben. Je nachdem, ob AktFarbeVorberechnet TRUE oder FALSE ist, setzt sich die Farbe eines Punktes wie folgt zusammen:

Tabelle 15.1 Größen für die Berechnung eines „weichen" Übergangs zwischen Radiosity-Berechung und Rendering

Bezeichnung	Bedeutung
C_P	letztlich dargestellte Farbe des Punktes P
C_{dB}	durch Beleuchtungsverfahren berechnete diffuse Farbgewichtung
g_d	Anteil der vorberechneten, diffusen Farbe (`AktDiffVorZuDiff`)
C_{dV}	durch Radiosity vorberechnete, diffuse Farbgewichtung (=einfallende Energiedichte, siehe Kap. 18.4)
C_{aV}	durch Radiosity vorberechnete, ambiente Farbgewichtung (=unverteilte mittlere Energiedichte)
R_A	diffuse Reflektivität der Fläche, die schattiert werden soll (siehe `GlobalerFlaechenTYP.Erw.Reflektivitaet` in Anhang A.9)
C_{sB}	durch Beleuchtungsverfahren berechnete, spekulare Farbe (siehe Abschnitt 12)
g_s	Dämpfung des spekularen Anteils (`AktSpekGewichtung`)

Wenn `AktFarbeVorberechnet` erfüllt ist:

$$C_P = (C_{dB} \cdot (1 - g_d) + C_{dV} \cdot g_d + C_{aV}) \cdot R_A + C_{sB} \cdot g_s,$$

und wenn `AktFarbeVorberechnet` nicht erfüllt ist:

$$C_P = C_{dB} \cdot R_A + C_{sB}.$$

Hierbei gelten die in Tabelle 15.1 angegebenen Abkürzungen. Mit diesen Variablen kann also weich zwischen Radiosity-Beleuchtung und „normaler" Beleuchtung gemischt werden.

Variablen mit gleicher Bedeutung werden übrigens auch in der Unit `PRIMREND` verwendet. Für alle Punkte, die nicht scharf sind, erfolgt nämlich das Beleuchten in diesem Modul, da für solche Punkte eine Beleuchtungsberechnung unabhängig von den zugehörigen Flächen vorgenommen werden kann.

15.4.2 Die Prozedur `StelleFlaecheDar`

```
PROCEDURE StelleFlaecheDar(VAR Flaeche: GlobaleFlaecheTYP);
```

Nachdem einmal für alle Flächen gemeinsam `SetzeSchattierungsParameter` aufgerufen wurde, gelangen diese Flächen mittels des Aufrufes von `StelleFlaecheDar` zur Darstellung. Der Typ `GlobaleFlaecheTYP` wird in der Beschreibung der globalen Parameter `PRIMGLOB` in voller Länge aufgeführt (siehe Anhang A.9.1.10). Hier nur die für die Schattierung relevanten Einträge:

```
GlobaleFlaecheTYP = RECORD
                      ...
                      Erw.Normale        : Vektor3DTYP;
                      Erw.PunktAnzahl    : BYTE;
                      Erw.Reflektivitaet : FarbVektorTYP;
                      EigenschaftsRef    : EigenschaftPtrTYP;
                      PunktIndizes       : ARRAY[0..3] OF LONGINT;
                      ...
                    END;
```

`Normale` ist der Normalenvektor der Fläche. Er wird beim Beleuchten von Eckpunkten benutzt, wenn diese scharf sind. Über die `PunktAnzahl` kann entschieden werden, ob es sich bei der darzustellenden Fläche um Drei- oder Vierecke handelt. Andere N-Ecke sind an dieser Stelle nicht möglich, da sie vorher (i.a. direkt beim Einlesen) zerteilt wurden.

`Reflektivitaet` wurde schon im Zusammenhang mit `SetzeSchattierungsParameter` angesprochen. Die diffuse Farbgewichtung multipliziert mit der `Reflektivitaet` ergibt die diffuse Farbe der Fläche nach der Beleuchtung. `EigenschaftsRef` zeigt auf eine Datenstruktur, in der die Eigenschaften von Teilobjekten gespeichert sind, die sich nicht von Fläche zu Fläche ändern.

Die für die Schattierung relevanten Teile von `EigenschaftTYP` sind:

```
EigenschaftTYP  = RECORD
                    ...
                    T                  : SINGLE;
                    ArtDerSchattierung: ArtDerSchattierungTYP;
                    Textur             : TexturTYP;
                    TexZSkalierung     : SINGLE;
                    ...
                  END;
```

Dabei steht `T` für Transparenz, wobei 0 vollkommen undurchsichtig und 1 vollkommen durchsichtig entspricht. `ArtDerSchattierung` kann die Werte `(Flach, Gouraud, Phong)` annehmen. In `Textur` ist gespeichert, ob und wenn ja, wie ein Teilobjekt texturiert ist. Als Art der Textur sind die Werte `(Keine, Textur2d, TexturBump, TexturSolid)` möglich. Für `Textur2d` und `TexturBump` wird über einen Dateinamen auf die Textur zugegriffen und zwar mit den Routinen `LadeTextur` und `LeseTexturPixel` der Unit `PRIMMEM` (Anhang C). Dagegen generieren bei der Texturart `TexturSolid` Prozeduren den Wert der Textur.

`TexZSkalierung` dient beim Bumpmapping zur Skalierung der „Höhe" der Textur. Der Wert entsteht aufgrund der Texturtransformationen und wird mit der Höhe einer Bump-Textur multipliziert, bevor der daraus resultierende Normalenvektor berechnet wird.

Als letzter Eintrag von `GlobaleFlaecheTYP`, der in `PRIMSHD` benötigt wird, muß noch `PunktIndizes` erläutert werden. Wie der Name schon sagt, gibt `PunktIndizes` die Indizes der (Eck-)Punkte an, aus denen sich die Fläche bildet. Mittels dieser Indizes können die Punkte geladen und gespeichert werden. Ein Punkt selbst ist ein Record, von dem das Schattieren folgende Komponenten benutzt:

```
PunktTYP   = RECORD
               Koordinate          : Vektor3DTYP;
               Normale             : Vektor3DTYP;
               TexturKoordinate    : Vektor3DTYP;
               KoordinateDC        : Vektor3DTYP;
               KoordinateNPC       : Vektor4DTYP;
               ClipCode            : BYTE;
               IstScharf           : BOOLEAN;
               DiffFarbe,SpekFarbe : FarbVektorTYP;
               ...
             END;
```

`Koordinate` entspricht den Weltkoordinaten des Punktes. Diese werden beim Schattieren zum Berechnen der partiellen Ableitungen der `TexturKoordinaten` für das Bumpmapping benötigt. Das Phong-Shading benutzt die `Normale`. `Texturkoordinate` findet bei texturierten Flächen Verwendung. `KoordinateDC` legt die Position des Eckpunktes in Pixelkoordinaten fest. Diese Position entspricht der oben angesprochenen Subpixelposition der Eckpunkte.

`KoordinateNPC` sowie `ClipCode` werden bei Flächen, die teilweise außerhalb des Sichtbereiches liegen, verwendet. `ClipCode` hat für Punkte, die innerhalb des Sichtbereiches liegen, beim Aufruf von `StelleFlaecheDar` den Wert 0. Wenn `IstScharf` erfüllt ist, so wird beim Phong-Shading der Normalenvektor der Fläche, nicht der des Punktes, als Anfangswert für die Normaleninterpolation verwendet.

In `DiffFarbe` und `SpekFarbe` sind die Informationen, die anhand der Beleuchtungsberechnung gewonnen wurden, gespeichert. Diese beiden Werte sind beim Schattieren für alle nicht scharfen Punkte, die innerhalb des Sichtbereiches liegen, also den `ClipCode` 0 haben, bereits berechnet. Im anderen Fall enthält `DiffFarbe` die vorbeleuchtete Farbe des Punktes, wenn eine Radiosity-Iteration stattgefunden hat, sonst ist er undefiniert.

Nach dieser langen Vorrede über Typen, die von `StelleFlaecheDar` benutzt werden, nun zur eigentlichen Implementierung. Um zu entscheiden, ob die Fläche ganz innerhalb, ganz außerhalb oder teilweise innerhalb des Sichtbereiches liegt, werden die Clipcodes aller Eckpunkte logisch miteinander verknüpft. Wenn das Ergebnis dieser Verknüpfung anzeigt, daß die Fläche ganz außerhalb des Sichtbereiches liegt, wird die Routine auf kürzestem Weg via `exit` verlassen. Anderenfalls weist die Prozedur der Boole'schen Variable `toClip` den Wert `TRUE` zu, wenn die Fläche im späteren Verlauf zu clippen ist.

Als zweites erfolgt eine Beleuchtung der scharfen oder außerhalb des Sichtbereiches liegenden Punkte der Fläche. Anschließend werden alle diffusen Punktfarben mit der Reflektivität der Fläche multipliziert, so daß nach diesem Teil die Eckpunktfarben der darzustellenden Fläche korrekt berechnet sind.

Das Prinzip für den nächsten Schritt von `StelleFlaecheDar` ist im Abschnitt 15.1 bereits angesprochen worden: Je nach Art der Schattierung und Texturierung werden verschiedene Initialisierungen vorgenommen und die Pixelfunktion, die während der Rasterisierung das eigentliche Pixelsetzen übernimmt, eingesetzt. Der Name dieser Pixelfunktion ist `BerechnePixelProc`.

Damit ergibt sich folgender Ablauf von `StelleFlaecheDar`:

Algorithmus 15.9 (Ablauf von `StelleFlaecheDar`)

ClipCodeIn=0, ClipCodeOut=11111111$_2$.
FÜR alle Punkte der Fläche:
ClipCodeIn ← ClipCodeIn bitweise ODER mit Punkt.ClipCode.
ClipCodeOut ← ClipCodeOut bitweise UND mit Punkt.ClipCode.
ENDE FÜR alle Punkte der Fläche.
WENN ClipCodeIn=0, DANN:
ToClip := FALSE.
SONST:
WENN ClipCodeOut=0, DANN:
ToClip := TRUE
SONST:
Ende der Routine, da die Fläche komplett außerhalb ist.
FÜR alle Punkte:
WENN der Punkt scharf ist, DANN:
Punkt.Normale := Fläche.Normale.
WENN der Punkt scharf oder der Clipcode ≠ 0 ist, DANN:
Beleuchte den Punkt.
Multipliziere die diffuse Farbe des Punktes mit der ..
.. Reflektivität der Fläche.
ENDE FÜR alle Punkte.
Bestimme in Abhängigkeit der Schattierungsart und der Art der ..
.. Textur die Anzahl der Parameter, die während der ..
.. Rasterisierung interpoliert werden müssen.
Initialisiere die Werte, die beim Pixelsetzen verwendet werden, ..
.. während der Rasterisierung aber konstant sind.
(Beim Flat-Shading ist dies z.B. der Mittelwert der Eckpunktfarben.)
Setze die Pixelsetzfunktion in Abhängigkeit von der Schattierungsart.
WENN ToClip = TRUE, DANN:
Clippe die Fläche.
(Die Clippingroutine ruft anschließend die ..
.. Rasterisierungsroutine selbsttätig auf)
SONST:
WENN Fläche ein Viereck, DANN:
Rasterisiere zwei Dreiecke.
SONST:
Rasterisiere ein Dreieck.
ENDE WENN Fläche ein Viereck, SONST.
ENDE WENN ToClip = TRUE, SONST.

Was die Pixelsetzroutinen im einzelnen ausführen, war bereits Gegenstand des Abschnittes 15.1. Bei texturierten Flächen ist es noch nötig, die Ergebnisfarbe mit der Texturfarbe zu modulieren. Hierzu liest die Pixelfunktion bei 2D-Texturen und beim Bumpmapping den Texturwert aus dem Speicher. Bei den 3D-Texturen wendet sie die Texturprozedur an, um die Texturfarbe zu bestimmen.

Wie wird nun die Textur in die Berechnung der Pixelfarbe einbezogen? Beim Bumpmapping ist die Antwort eindeutig: Die Textur moduliert den Normalenvektor *vor* der Beleuchtung. Die Schattierungsart beim Bumpmapping ist immer Phong. Bei 2D- und 3D-Texturen hängt das Mischungsverfahren von der Schattierungsart ab. Beim Flat- und Gouraud-Shading werden die Texturfarben mit der Flächenfarbe bzw. der interpolierten Farbe komponentenweise multipliziert. Beim Phong-Shading, bei dem nach der Beleuchtungsberechnung diffuse und spekulare Farbe getrennt vorhanden sind, ergibt sich die dargestellte Farbe zu

$$C = C_d \cdot C_T + C_s$$

wobei C_d die mit Hilfe der Beleuchtungsroutine gewonnene diffuse Farbe, C_s die auf die gleiche Art gewonnene spekulare Farbe und C_T die Farbe der Textur ist. Die Textur beeinflußt also beim Phong-Shading nicht die Farbe der Glanzlichter. Dies entspricht eher der Physik, als die diffuse und die spekulare Farbe zu modulieren[1].

Wenn die Transparenz der Fläche $\neq 0$ ist, muß ganz am Ende der Pixelroutine die Pixelfarbe noch mit der Hintergrundfarbe gemischt werden, wobei das Mischungsverhältnis durch die Transparenz der Fläche gegeben ist. Wir haben den α-Buffer für Transparenz nicht realisiert, da dieser auch nicht in jedem Fall funktioniert. Aus diesem Grund ist die dargestellte Transparenz mit Vorsicht zu genießen. Halbwegs richtige Ergebnisse sind nur dann erreichbar, wenn die transparenten Flächen nach den opaken, in ihrer Reihenfolge von „hinten" nach „vorne", dargestellt werden.

Bei der Radiosity-Iteration wird die Transparenz übrigens gar nicht berücksichtigt, so daß eine fast durchsichtige Fläche die gleichen Schatten verursacht wie eine undurchsichtige. Hier sei es Ihnen überlassen, das Programm entsprechend zu erweitern.

Im Gegensatz zu den Schattierungsverfahren, deren Implementierung in diesem Abschnitt erläutert wird, ist allerdings mit dem im Teil 4 vorgestellten Raytracing eine recht realistische Darstellung von transparenten Objekten gegeben. Dort ist es dann auch möglich, nicht nur die Hülle eines transparenten Objektes darzustellen, sondern aufgrund des Brechungsindex auch das Innere.

Zurück zum Ablauf von `StelleFlaecheDar`. Am Ende ruft die Prozedur ein- oder zweimal die Routine `ZeichneDreieck` auf. Diese übernimmt die eigentliche Rasterisierung der Fläche und die Interpolation der Werte im Inneren. Ob sie ein- oder zweimal aufgerufen wird, hängt davon ab, ob die Fläche ein Drei- oder Viereck ist.

[1]Das Phongsche Beleuchtungsmodell an sich ist rein empirisch, so daß es an dieser Stelle relativ schwer fällt, von physikalischer Entsprechung zu sprechen.

15.4.3 Die Prozedur ZeichneDreieck

Über ZeichneDreieck muß nicht mehr viel gesagt werden. Der Grundalgorithmus wurde im vorigen Abschnitt bereits näher erläutert. Dieser Algorithmus wird nun von einem Wert c auf ein Feld von Werten c_i erweitert und die Initialisierung dieser i Werte erfolgt ganz analog zur Initialisierung des einen Wertes im Grundalgorithmus. Damit die Pixelprozedur „weiß", welcher Wert c_i welche Bedeutung hat, sind diverse Konstanten definiert, die je nach Schattierungsart dem Index der zu interpolierenden Werte entsprechen. Die Konstanten z_i, g_r, g_g, ... sind direkt am Anfang des Implementierungsteils von PRIMSHD definiert.

Nachdem alle Eckpunktwerte des Dreiecks in das Festkommaformat konvertiert und in entsprechenden Variablen abgespeichert sind, erfolgt die eigentliche Rasterisierung in ZeichneDreieck. Das Setzen der Eckpunktwerte übernimmt die Routine SetzePunktWerte.

Abschließend noch einige Worte zum z-Buffervergleich: Da unabhängig von Schattierungsart und Textur immer ein z-Buffervergleich pro Pixel notwendig ist, geschieht dieser nicht in der Pixelprozedur, sondern direkt in der Dreiecksroutine. Wenn aufgrund des Vergleiches das Pixel nicht zu setzen ist, wird die Pixelroutine erst gar nicht aufgerufen. Da auf den z-Buffer nicht direkt, sondern nur mit Hilfe der Speicherverwaltungsroutinen (siehe Anhang C) zugegriffen werden kann, liest die Spanroutine zu Beginn alle Werte des z-Buffers für den zu zeichnenden Span und schreibt die aktualisierten z-Werte für den ganzen Span auf einmal zurück. Auf diese Art fällt der von der Speicherverwaltungsroutine produzierte „Overhead" nicht so stark ins Gewicht.

15.4.4 Die Prozedur ZeichneHintergrund

```
PROCEDURE ZeichneHintergrund(VAR Hintergrund: HintergrundTYP;
                             x, y, b, h: INTEGER);
```

Der Typ HintergrundTYP wird in Anhang A genau beschrieben.

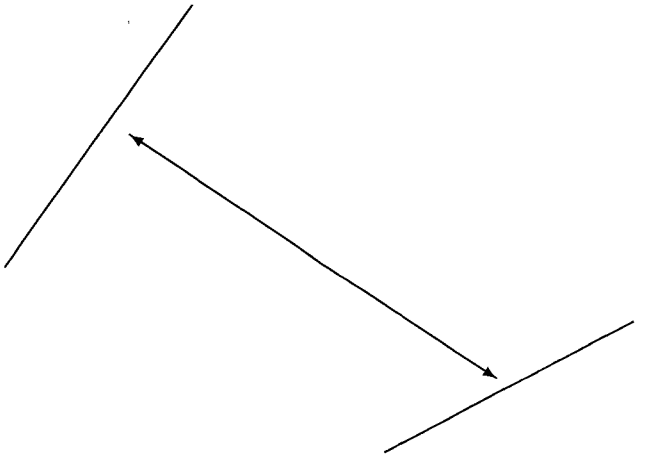
Falls als Hintergrund eine Textur definiert ist, dann werden in dem durch x, y, b und h gegebenen Fenster zwei texturierte Dreiecke dargestellt. Die diffuse Intensität wird dabei mit dem linear interpolierten Farbwert zwischen Hintergrund.FarbeOben und Hintergrund.FarbeUnten gewichtet. Die z-Werte der beiden Dreiecke werden auf den Maximalwert („weit weg") gesetzt.

Soll im Hintergrund keine Textur sichtbar sein, so wird lediglich die Farbe des Hintergrundes zwischen Hintergrund.FarbeOben und Hintergrund.FarbeUnten linear interpoliert.

III

Radiosity

Ute Claussen
Josef Pöpsel



16. Radiosity

Radiosity heißt eines der jüngsten Verfahren, mit dem in der Computergrafik realistische Bilder erzeugt werden (Abb. V bis X) [CP91].

Alle bisher genannten Verfahren haben einiges gemeinsam: Sie sind in ihrem Aufwand linear abhängig von der Zahl der Objekte, der Lichtquellen und der Pixel. Mit ihnen können in der ursprünglichen Form keine Schatten erzeugt werden. Sie berücksichtigen keine mehrfachen Reflexionen zwischen Objekten, d.h. sie berechnen nur lokale Beleuchtungseffekte. Aber: Sie geben einen einigermaßen realistischen dreidimensionalen Eindruck eines Objektes wieder und sind (relativ) „billig" in ihrer Realisierung.

Nachdem für das Gouraud- und das Phong-Shading mit verschiedenen Tricks versucht wird, z.B. Reflexionen durch Umgebungs-Mapping oder Schatten durch das Einführen von Schatten-Polygonen zu simulieren, kann dies durch sogenannte globale Beleuchtungsberechnungen in einem einzigen Verfahren erzeugt werden. Diese sind das Strahlverfolgungs- (Raytracing-) und das Strahlungs- oder Radiosity-Verfahren. Zuerst ein kleiner Exkurs in die Strahlverfolgung, der ja im Anschluß ein ganzer Teil dieses Buches gewidmet ist.

In einem klassischen Artikel beschrieb Turner Whitted 1980 das Prinzip, das uns heute so einleuchtend vorkommt: Ausgehend vom Blickpunkt einer Szene wird durch jedes Pixel des Bildschirms ein Strahl „geschossen" [Whi80]. Trifft dieser Strahl kein Objekt, so ist die Farbe des Pixels z.B. eine vordefinierte Hintergrundfarbe. Trifft dieser Strahl aber ein Objekt, so setzt sich die wahrgenommene Farbe aus drei Anteilen zusammen. Der ambiente Anteil kann direkt berechnet werden. Für den diffusen Anteil muß festgestellt werden, welche Lichtquellen das Objekt „sieht". Der spiegelnde Anteil entspricht – gemäß dem Reflexions- und dem Brechungsgesetz – den Intensitäten, die aus der gespiegelten bzw. der gebrochenen Strahlrichtung auf das Objekt auftreffen. Das bedeutet also, daß diese Intensitäten nun rekursiv berechnet werden müssen. Die Rekursion endet, wenn ein Strahl die Szene verläßt, eine Lichtquelle trifft oder wenn die mit dem Strahl „transportierte" Intensität einen bestimmten Wert unterschreitet.

Mit dem Prinzip der Strahlverfolgung war das erste Mal ein globales Beleuchtungsmodell beschrieben. Mit dem Algorithmus können Schatten erzeugt werden und er ist aufgrund der Modellierung des Brechungs- und des Reflexionsgesetzes imstande, ideal spiegelnde Objekte perfekt darzustellen. Deswegen handelt es sich bei den Bildern, die zur Demonstration erzeugt werden, auch

meist um spiegelnde Kugeln, denn für Kugeln läßt sich auch der Schnittpunkt mit dem Strahl besonders einfach berechnen.

So einfach sich das Verfahren beschreiben läßt und so viele der vorher nicht möglichen Effekte damit modelliert werden können, es bleiben doch einige Haken: Der bei weitem schwerwiegendste Nachteil ist der hohe Rechenaufwand. Ein zweiter Nachteil ist die Abhängigkeit vom Blickpunkt. Da in den meisten Anwendungen zunehmend Animationen, also Sequenzen von Bildern, eingesetzt werden, muß jedes Bild von neuem berechnet werden. Weiterhin treten Abtastfehler auf, denen man durch mehrfaches Schießen eines Strahles durch ein Pixel versucht, zu begegnen. Dies erhöht zusätzlich den Berechnungsaufwand.

Nicht zuletzt ist zwar im Zusammenhang mit Raytracing-Bildern erstmals von „Fotorealismus" gesprochen worden, die Bilder und Objekte und auch die gestochen scharfen Schatten haben aber nichts mit der beobachtbaren Realität zu tun. Schaut man sich in der Umwelt um, so sieht man hauptsächlich diffus reflektierende Flächen. Aus diesem Grund haben die Schatten „verwischte" Kanten: Die Flächen werden indirekt beleuchtet (Abb. V).

Einen Ansatz, diese Merkmale der Realität in einem globalen Beleuchtungsmodell abzubilden, existiert seit 1984 mit dem sogenannten „Radiosity-Verfahren" [GTGB84]. Grundlage des Verfahrens bilden die Hauptsätze der Thermodynamik. Sie machen Aussagen über das energetische Verhalten geschlossener Systeme. Der erste Hauptsatz ist der von Robert Mayer formulierte Energieerhaltungssatz:

> *Die Energiesumme in einem abgeschlossenen System, dem also weder Energie zugeführt noch entzogen wird, ist konstant.*

Dementsprechend gehört zu jedem Zustand eines Systems ein eindeutig bestimmter Wert der inneren Energie. Dieser Satz ist die wichtigste Grundlage des Strahlungs- oder Radiosity-Verfahrens. Der zweite Hauptsatz besagt, daß alle Naturvorgänge so verlaufen, daß die gesamte Entropie aller beteiligten Körper zunimmt. Als Schlußfolgerung für ein geschlossenes System gilt, daß in diesen die Entropie niemals abnimmt.

Licht ist eine Form von Energie. Daher können die Sätze der Thermodynamik direkt für die Simulation von Beleuchtungsverhältnissen verwendet werden. Im folgenden gelten daher einige Voraussetzungen:

1. Alle Flächen in einer Szene sind ideal diffus reflektierende Flächen.
2. Alle Flächen sind homogen bezüglich ihrer Eigenschaften, haben also z.B. über die Fläche eine konstante Farbe.
3. Die Szene ist abgeschlossen. Das bedeutet, daß der Szene weder Energie zugeführt noch entzogen wird.

Die von einer Fläche abgestrahlte Energie setzt sich aus der eigenen Energie (falls sie eine Lichtquelle ist) und der gewichteten Summe aller auf diese Fläche auftreffenden Energien von anderen Flächen zusammen (siehe Abb. 16.1).

Als Formel ausgedrückt:

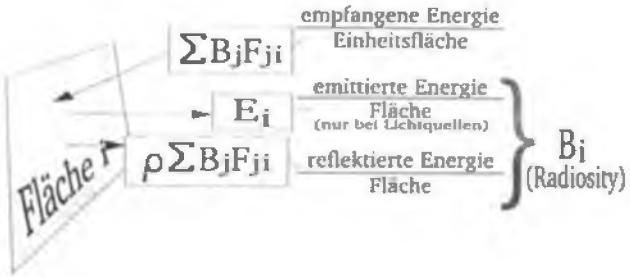


Abb. 16.1 Die Radiosity einer Fläche ist die Energie, die diese Fläche verläßt, bezogen auf den Flächeninhalt

Tabelle 16.1 Erläuterung der für das Radiosity-Verfahren benutzten Notation

Bezeichnung	Bedeutung
B_i	Energiedichte (also Energie pro Zeit- und Flächeneinheit) der Fläche i, auch Radiosity der Fläche i genannt.
E_i	Energiedichte der Fläche i ohne Fremdeinwirkung (Emmission).
ρ_i	Reflektivität der Fläche i. Sie gibt an, welcher Teil des einfallenden Lichtes auch wieder abgestrahlt wird und ist wellenlängenabhängig.
F_{ji}	Der sogenannte Formfaktor. Er gibt an, welcher Teil der Energiedichte einer Fläche j auf die Fläche i übergeht und ist allein von der Geometrie einer Szene abhängig.
N	Die Anzahl der Flächen in der Szene.

$$1 \leq i \leq N: \quad B_i = E_i + \rho_i \sum_{j=1}^{N} B_j F_{ij}$$

Dabei gelten die in Tabelle 16.1 erklärten Bedeutungen. Dies sind N Gleichungen für N unbekannte Größen, nämlich die B_1 bis B_N. Also handelt es sich um ein lineares Gleichungssystem der Form

$$A \cdot B = E$$

wobei

$$A = \begin{pmatrix} 1-\rho_1 F_{11} & -\rho_1 F_{12} & \cdots & -\rho_1 F_{1N} \\ -\rho_2 F_{21} & 1-\rho_2 F_{22} & \cdots & -\rho_2 F_{2N} \\ \vdots & \vdots & & \vdots \\ -\rho_N F_{N1} & -\rho_N F_{N2} & \cdots & 1-\rho_N F_{NN} \end{pmatrix},$$

$$B = \begin{pmatrix} B_1 \\ B_2 \\ \vdots \\ B_N \end{pmatrix} \quad \text{und} \quad E = \begin{pmatrix} E_1 \\ E_2 \\ \vdots \\ E_N \end{pmatrix}$$

sind. Die Lösung B_1 bis B_N gibt die Energiedichte auf jeder Fläche an. Die auf dem Monitor dargestellte Intensität einer Szene ist proportional zu dieser Energiedichte.

Bevor wir zu weiteren Einzelheiten des Verfahrens kommen, soll die Radiosity-Methode hier kurz mit der Strahlverfolgung verglichen werden. Letztere ist optimal zur Darstellung spiegelnder Oberflächen. Dagegen ist das Radiosity-Verfahren optimal zur Darstellung völlig diffuser Umgebungen. Es wird daher auch oft für den Innenarchitekturbereich angewendet, so wie wir das hier in einem Beispiel mit der Küche auch getan haben (Abb. V). Der Aufwand zur Berechnung kann mit einem iterativen Lösungsverfahren so reduziert werden, daß von interaktiver Anwendung gesprochen werden kann.

Ein weiterer Vorteil des Radiosity-Verfahrens liegt darin, daß seine Lösung blickpunktunabhängig ist. Ist also die Lichtverteilung in einer Szene erst einmal berechnet, so führen Änderungen des Blickpunktes nicht zu einem neuen Gleichungssystem. Dies ist natürlich insbesondere für Kamera-Animationen von Vorteil, in denen man sich durch die vorher berechnete Szene bewegt. Erst wenn sich die Geometrie, die Lichtquellen oder die Reflexionseigenschaften von Objekten ändern, muß ein neues Gleichungssystem gelöst werden.

Leider existieren aber weder nur ideal diffuse noch nur ideal spiegelnde Szenen bzw. Objekte. Ein Ansatz, die Methoden der Strahlverfolgung und der Strahlungsberechnung zu kombinieren, ist als Mehrpassverfahren bekannt geworden. Er wird im Laufe dieses Kapitels nochmals aufgegriffen.

Im Anschluß werden die wesentlichen Bestandteile des Radiosity-Algorithmus beschrieben: die Aufstellung des Gleichungssystems und dessen Lösung. Die Aufstellung entspricht im wesentlichen der Berechnung der Formfaktoren.

16.1 Berechnung der Formfaktoren

Die Berechnung der Formfaktoren ist der beim Radiosity-Verfahren bei weitem aufwendigste Teil. Beschreibt man die Formfaktoren für endliche Flächen i und j in einer konvexen Umgebung, also wenn keine Objekte sich gegenseitig verdecken, so gilt:

$$1 \leq i,j \leq N: \quad F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\phi_i cos\phi_j}{\pi r^2} dA_j dA_i$$

Die Bedeutung der einzelnen Größen können Sie Abb. 16.2 entnehmen. A_i bzw. A_j bezeichnen die Flächen, r den Abstand der Flächen zueinander, ϕ_i bzw. ϕ_j den Winkel zwischen der Flächennormale und dem Richtungsvektor zur anderen Fläche. Wird die Forderung nach Konvexität aufgehoben, können die Flächen gegenseitig verdeckt sein. Dem kann durch Einführen einer Größe h_{ij} unter dem Integral Rechnung getragen werden, die genau dies angibt.

$$1 \leq i,j \leq N: \quad F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{cos\phi_i cos\phi_j}{\pi r^2} h_{ij} dA_j dA_i$$

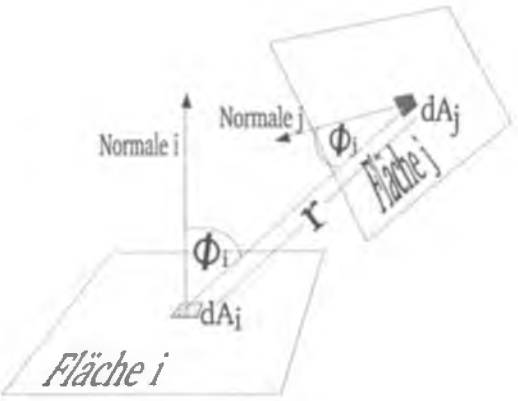


Abb. 16.2 Geometrie der Formfaktorberechnung

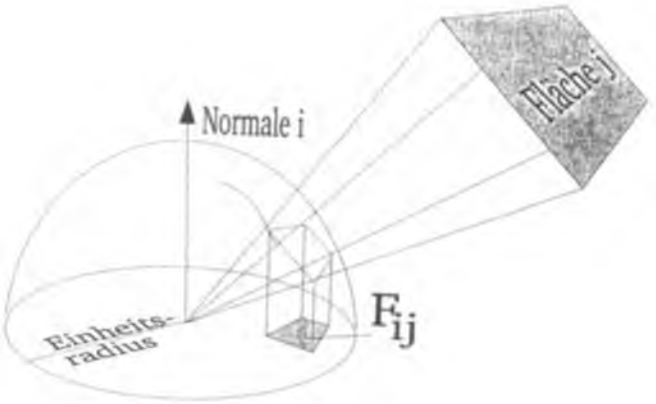


Abb. 16.3 Geometrie des Nusselt-Analogon

Die exakte Berechnung dieses Integrals stellt sich als ziemlich schwierig heraus. Es ist nur für bestimmte Flächen als Integral geschlossen lösbar. Für kompliziertere Flächen muß es numerisch gelöst werden. Daher sucht man nach alternativen Berechnungsmethoden, die dieses Integral ausreichend genau annähern und bedient sich unter anderem der Analogie von Nusselt (Abb. 16.3). Sie besagt, daß der Formfaktor F_{ij} dem Anteil der projizierten Fläche A_j an der Grundfläche einer Einheitshalbkugel um einen Punkt auf A_i entspricht. Dabei wird A_j zuerst auf die Oberfläche der Halbkugel und anschließend orthogonal auf die Fläche A_i projiziert.

Auch diese Berechnung ist, insbesondere unter Miteinbeziehung des Verdeckungsproblems, analytisch immer noch sehr schwer zu beschreiben. Die Einheitshalbkugel wird daher durch einen „Einheitshalbwürfel" ersetzt [CG85]. Auf diesen Einheitshalbwürfel („hemi cube") kann nun wesentlich einfacher eine Projektion berechnet werden. Rastert man nun noch diesen Würfel in „Pixel", so kann die Berechnung der Formfaktoren mittels einer Verdeckungsrechnung auf

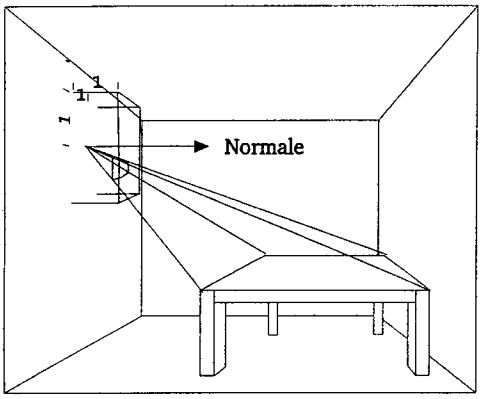


Abb. 16.4 Geometrie der Formfaktorberechnung mit einem Halbwürfel

den fünf „Mini-Bildschirmen" des Würfels durchgeführt werden. Zu jedem der Pixel wird ein ΔF_{ij} in einer Tabelle abgelegt. Ist die Szene auf dem Würfel dargestellt, müssen noch alle ΔF_{ij}, die von einer Fläche bedeckt sind, addiert werden, um den Formfaktor F_{ij} zu erhalten. Mit einem Halbwürfel werden also die Formfaktoren für alle Flächen j und ein festes i bestimmt. Zur Berechnung aller Formfaktoren muß die Szene jeweils auf dem Halbwürfel über jeder der N Flächen dargestellt werden (Abb. 16.4).

Durch die Abtastung des Raumes entstehen Fehler, sogenannte Alias-Effekte. Diesen kann durch erhöhte Auflösung des Würfels entgegengewirkt werden. Sinnvoll ist das vor allem dort, wo sich die Fehler am stärksten bemerkbar machen, nämlich in der Richtung des Normalenvektors. Eine weitere Möglichkeit, die Fehler zumindest zu streuen, besteht darin, den Würfel zufällig um den Normalenvektor zu rotieren.

Zusätzlich läßt sich die Berechnung der Formfaktoren vereinfachen, indem nicht mehr ein Halbwürfel, sondern nur noch der „Deckel" des Würfels, also eine Ebene verwendet wird (Abb. 16.5). Je nach Abstand und Größe der Ebene verliert man zwar einen Teil der Szeneninformation, aber der Rechenaufwand reduziert sich doch erheblich, da es nur einmal pro Fläche notwendig ist, die Szene darzustellen. In der ursprünglich vorgeschlagenen Form ist die Ebene zum Normalenvektor hin stetig feiner unterteilt.

Zum Schluß seien noch ein paar hilfreiche Beziehungen angegeben, die den Speicheraufwand bei einer Implementierung reduzieren helfen.

Die Beziehung zwischen den Flächen A_i und A_j ist symmetrisch, daher gilt:

$$1 \leq i, j \leq N: \quad F_{ji} = \frac{A_i}{A_j} F_{ij}.$$

Weiterhin verschwinden die Formfaktoren F_{ii}, sofern es sich um planare Flächen A_i handelt.

$$1 \leq i \leq N: \quad F_{ii} = 0$$

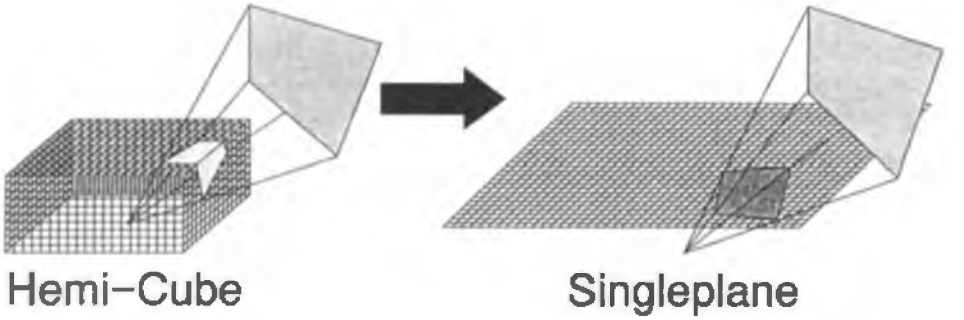


Abb. 16.5 Reduzierung des Halbwürfels auf eine Ebene

Und zu guter letzt ist aufgrund der Definition der Formfaktoren und aufgrund der Energieerhaltung die Summe aller Formfaktoren einer Fläche gleich 1.

$$1 \leq j \leq N: \quad \sum_{i=1}^{N} F_{ji} = 1.$$

Damit sind die Formfaktoren berechnet. Der nächste Abschnitt befaßt sich mit den Lösungsmöglichkeiten für das Gleichungssystem.

16.2 Lösung des Gleichungssystems

Nach der Formfaktorberechnung muß das Gleichungssystem $A \cdot B = E$ nach B aufgelöst werden. Auch dazu gibt es mehrere Möglichkeiten. Eine erste ist die Lösung des Gleichungssystems durch Invertierung der Matrix A. Multipliziert man die Gleichung von links mit A^{-1}, so erhält man die Lösung:

$$B = A^{-1} \cdot E.$$

Sie ist als „full-matrix"-Lösung bekannt. Zu ihrer Bestimmung können Programme verwendet werden, die es fast in jeder Bibliothek gibt. Die Invertierung der Matrix A bereitet im Prinzip keine Probleme. Sie ist aber eine Lösung mit Brachialgewalt, insbesondere, wenn man den Speicherplatz von N^2 Matrixelementen und die Rechenzeit in der Größenordnung von N^3 bedenkt. Schon Szenen mittlerer Komplexität haben heute zwischen mehreren tausend und zigtausend Flächen. Natürlich gibt es zur Lösung dieser Gleichung auch iterative Methoden. Die Matrix A ist aufgrund ihrer Diagonaldominanz auch dafür geeignet.

Verblüffenderweise existiert für dieses Iterationsverfahren auch eine andere, energieorientierte Interpretation, die recht einleuchtend ist. Die Vorgehensweise ist folgende: Es wird für jeden Schritt des Verfahrens die Fläche j bestimmt, die

am meisten Energie zu „verteilen" hat. Das werden am Anfang die Lichtquellen sein. Für diese eine Fläche werden alle Formfaktoren F_{ij} bestimmt, d.h. ein Einheitshalbwürfel wird ausgewertet. Anschließend wird die vorhandene Energie gemäß der Formfaktoren zu den Energien der anderen Flächen addiert. Danach wird die nächste Fläche gesucht, die nun noch die meiste Energie zu verteilen hat.

Es kann gezeigt werden, daß dieses Verfahren gegen die Lösung B von oben konvergiert. Verteilt man während des Verfahrens noch zusätzlich die noch nicht verteilte Gesamtenergie zu Ausgabezwecken als ambienten Term gleichmäßig auf alle Flächen, so kann bereits nach einem Iterationsschritt ein sinnvolles Bild dargestellt werden. Dieses entwickelt sich im Laufe der Iterationen hin zur genauen Lösung. Das Verfahren ist als „progressive radiosity" bekannt und wurde 1988 erstmals veröffentlicht [CCWG88].

16.3 Mehrpassverfahren

Mehrpassverfahren basieren auf der Idee, daß die einzelnen Anteile des Lichts, also der diffuse und der spiegelnde, durch Überlagerung verrechnet werden können. Das Radiosity-Verfahren ist ideal für die Darstellung diffuser Körper und blickpunktunabhängig. Das Strahlverfolgungsverfahren dagegen ist ideal für die Darstellung spiegelnder Körper und abhängig vom Blickpunkt. Die Grundidee der Mehrpassverfahren ist, zuerst ein Radiosity-Verfahren und anschließend ein Raytracing-Verfahren durchzuführen. Allerdings werden durch die einfache Addition der Ergebnisse der beiden Verfahren nicht alle Effekte aufgenommen, die auftreten können. Beispielsweise werden weder mit dem einen noch mit dem anderen Verfahren Beleuchtungen von einem spiegelnden auf einen diffus reflektierenden Körper berechnet.

Insgesamt müssen also vier Effekte nachgebildet werden können: der Energietransport von einer diffusen bzw. einer spiegelnden auf eine diffuse bzw. spiegelnde Fläche. Der erste Vorschlag für ein solches Verfahren stammt von Wallace, Cohen und Greenberg [WCG87]. Für perfekt spiegelnde Flächen wird bei ihnen zusätzlich eine „reflektierte Welt" eingeführt, so daß im ersten Teil bereits spiegelnde Flächen mit berücksichtigt werden. Der zweite Teil des Verfahrens ist eine Erweiterung der Strahlverfolgung, so daß um die reflektierte Strahlrichtung herum die Intensitäten mit Hilfe eines Buffers integriert werden. Dadurch wird auch im zweiten Schritt sowohl diffuse als auch spiegelnde Reflexion oder Transmission modelliert.

17. Beispiele zur Radiosity-Berechnung

Anhand einer der simpelsten Szenen für die Radiosity-Berechnung sollen Sie in diesem Kapitel die Wirkung des Verfahrens und die Parameter des Programms kennenlernen. Mit Hilfe der PUNKTE- und FLAECHEN-Angabe wird ein Kasten modelliert, in den der Blick fällt. Die linke Wand ist grün, die hintere blau, die rechte rot, so wie in Abb. X des farbigen Bildteils.

Statt der Tori und der Kugel soll hier der Einfachheit halber eine Testfläche in die Szene gesetzt werden. Die Eingabe lautet wie folgt:

```
GED_VERSION 1;

OBJEKT wand;
  PUNKTE
    1.2, 1.2,-1.2;
    -1.2, 1.2,-1.2;
    -1.2,-1.2,-1.2;
    1.2,-1.2,-1.2;
  FLAECHEN
    0,1,2,3;
ENDOBJEKT;

BILDBREITE 4.5;
AUGE 0.1,-10,3;
REFERENZPUNKT 0,0,-0.2;
BILDABSTAND 4;
VORDERCLIPEBENE 2;
HINTERCLIPEBENE -2;

SZENE Beispiel10;
  KD 0.9; KA 0.0; KS 0.0;
  SCHATTIERUNG GOURAUD;
  OD 0.8,0.8,0.8;

  // Die diffuse Lichtquelle der Szene
  PUNKTE
    1.1,-0.05,-0.15;
    1.1,-0.05,-0.25;
    1.1,0.05,-0.25;
    1.1,0.05,-0.15;
```

```
  FLAECHEN
    0,3,2,1 B 600,600,600;

  // Die Waende des Wuerfels
  PUSH;
    OD 0.9,0.9,0.9; ZEICHNE wand;
    DREHE X,180;
    OD 0.9,0.9,0.9; ZEICHNE wand;
  POP;
  PUSH;
    DREHE Y,90;
    OD 0.3,0.9,0.3; ZEICHNE wand;
    DREHE X,180;
    OD 0.9,0.3,0.3; ZEICHNE wand;
  POP;
  PUSH;
    DREHE X,90;
    OD 0.4,0.4,0.9; ZEICHNE wand;
    DREHE X,180;
    OD 0.9,0.4,0.4; ZEICHNE wand;
  POP;

  // Die Testflaeche
  PUNKTE
    -0.5,-0.5,-0.7;
    -0.5, 0.5,-0.7;
     0.5,-0.5,-0.7;
     0.5, 0.5,-0.7;
  FLAECHEN
    0,1,3,2 FLIP;
END.
```

Wenn nun das Programm PRIM aufgerufen wird, geben Sie wie gewohnt Ihre Grafikkarte an. Die Nachfrage nach Patches und Elementen hat für die Anwendung des Radiosity-Verfahrens neue Auswirkungen. Daher sollten Sie diese Frage zuerst verneinen. Die Szene Beispiel10 bleibt daher so facettiert, wie Sie definiert wurde, d.h. sie enthält 9 Flächen, die an dem wechselseitigen Energieaustausch beteiligt werden.

Das Programm berechnet den Energieaustausch, wenn Sie die Frage nach der Radiosity-Berechnung bejahen. Wenn Sie auf die weiteren Fragen mit NEIN bzw. RETURN antworten, so beginnt das iterative Radiosity-Verfahren. Nach 8 Iterationen wird die beleuchtete Szene dargestellt, die an eine flach schattierte Darstellung erinnert.

Mit einer beliebigen Taste können die Iterationen abgebrochen werden. Der nächste Tastendruck nach der Darstellung des Bildes führt dann in das Menu von PRIM, aus dem mit

```
(I)teration fortsetzen
```

die Berechnung weitergeführt werden kann.

Das Aussehen dieses Beispiels, das nun beendet werden kann, ist auf die zweite Grundannahme des Radiosity-Verfahrens zurückzuführen: alle Flächen sind homogen bezüglich ihrer Eigenschaften, insbesondere also in bezug auf die Farbe bzw. Energiedichte. Es gibt also genau einen Energiewert pro definierter Fläche, in unserem Beispiel also 9. Große Flächen müssen also unterteilt werden, um diese Homogenitätsbedingung nicht zu verletzen und dennoch gut schattierte Ergebnisse zu erhalten.

Ein erste Unterteilung von Flächen sind die sogenannten Patches: Flächenstücke, die die richtige Größe zum „Verteilen" von Energie haben. Diese Patches unterteilen sich nochmals in Elemente, die Energie empfangen. Genaueres dazu können Sie in Abschnitt 18.4 oder auch bei Foley et.al. nachlesen [FvDFH90].

Lassen Sie nun nochmals das Beispiel laufen und geben auf die Frage nach der Unterteilung `JA` und für die Patch- und Elementgrößen die vorgesehenen Werte an. Die nachfolgende Meldung lautet nun, daß die Szene in 1900 Punkte, 393 Patches und 1644 Elemente, die nicht gleichzeitig Patch sind, unterteilt wurde. Nach 8 Iterationen wird das Bild angezeigt, auf dem jetzt bis auf die rechte Seitenfläche des Kastens alle Flächen bereits einen Teil Energie empfangen haben. Der Schatten der Testfläche auf dem Boden und an der linken Seitenwand ist deutlich erkennbar. Unterbrechen Sie die Iteration, so können Sie jeden der bereits diskutierten Parameter, z.B. den Blickpunkt, verstellen oder die Iterationen fortsetzen lassen.

Sollten in einer Szene also zu große Unterschiede in der Farbe von Flächen beim Darstellen auftreten, so können Sie die Patch- und Elementgröße auch kleiner wählen. Die Werte 0.6 und 0.2 haben sich bei Szenen mit einer maximalen Gesamtausdehnung von 2–3 als sinnvoll herausgestellt. Nach sehr vielen Iterationen wird in unserem Beispiel übrigens auch die rechte Seitenwand indirekt beleuchtet.

Mit zunehmender Anzahl von Iterationen wird das Bild heller, so wie auch in Abb. VII für ein größeres Beispiel dargestellt. Die Helligkeit, die sich bei der Iteration erst nach und nach ergibt, kann auch von Anfang an gleich mit dargestellt werden. Dieser sogenannte ambiente Anteil bewirkt, daß die Szene bereits ab der ersten Iteration hell wirkt, sich die eigentliche Lichtverteilung aber erst nach und nach einstellt. Bevor die Berechnung beginnt, ist für diese Art der Darstellung auf die Frage

```
Die unverteilte Energie als ambienten Anteil darstellen?
```

`JA` einzugeben. Informationen über die Größe des ambienten Anteils erhalten Sie direkt vor dieser Frage und bei Wahl dieser Option während der Iterationen. Am ambienten Anteil kann die Konvergenz des Verfahrens abgelesen werden.

Ein weiterer wichtiger Parameter für die Güte der berechneten Bilder ist die Auflösung der Formfaktor-Projektionsebene, der Singleplane. Dazu wollen wir ebenfalls ein Experiment durchführen. Starten Sie daher das Beispiel neu und unterteilen die Szene mit der Standardunterteilung. Wählen Sie `NEIN` oder die Defaultwerte für die restlichen Parameter. Die Auflösung der Singleplane wird auf 100 gesetzt. Zur Kontrolle könne Sie sich auch die Singleplane mit der projizierten Szene anzeigen lassen.

Stellt das Programm das Modell nun nach 8 Iterationen dar, so sind große Inhomogenitäten in der Schattierung der Szene zu erkennen. Dieser sogenannte „Streuselkuchen"-Effekt ist auf Abtastfehler zurückzuführen. Die Unterteilung der Szene ist so fein, daß einzelnen Flächen regelmäßig durch das Raster der Singleplanes „hindurchfallen" und daher keine Energie der Lichtquelle erhalten.

Dem kann durch zwei Maßnahmen abgeholfen werden: Der Erhöhung der Auflösung der Singleplane und der Einführung einer zweiten, kleineren Singleplane, die eine höhere Abtastung im „Inneren" der ersten Singleplane umsetzt. In der Beispielszene sollten Sie die Ergebnisse des eben durchgeführten Experimentes mit der Berechnung der Standardauflösung (300×300) vergleichen. Diese Auflösung führt nicht mehr zu den oben beobachteten Effekten, ebenso wie die Wahl der zweiten Singleplane mit der getesteten Auflösung von 100×100 Pixeln.

Mit weiteren Eingabemöglichkeiten vor Beginn der Iterationen werden die Breiten der Singleplanes oder die Anzahl der Iterationen bis zur nächsten Darstellung bestimmt und ob zusätzlich ein „normale", direkte Beleuchtung berechnet werden soll. Diese letzte Möglichkeit erlaubt die Kombination der Radiosity-Ergebnisse mit spiegelnden Effekten. Sie funktioniert nur, wenn Lichtquellen in der Szene definiert sind – in unserem Beispiel also nicht. Das Resultat können Sie aus Abb. VII ersehen.

18. Implementierung des Radiosity-Verfahrens

Die Beschreibung des Radiosity-Verfahrens erfolgt in drei Stufen. Zuerst suchen wir aus den theoretisch vorgestellten Verfahren diejenigen heraus, die uns geeignet erscheinen. Anschließend wird das Programm algorithmisch beschrieben und am Schluß die Realisierung in Pascal vorgestellt.

Der Ablauf eines Radiosity-Programms besteht aus vier Schritten (siehe Abb. 18.1):

1. Das Einlesen der Geometrie der Szene. Dafür wird die oben eingeführte Eingabesprache verwendet. Eine Einleseroutine setzt die Beschreibung der Geometrie in eine interne Datenstruktur um. Die interne Datenstruktur muß so angelegt sein, daß sie so viel Informationen wie nötig so effizient wie möglich speichert.
2. Die Berechnung der Formfaktoren. Hier muß entschieden werden, wie die Formfaktoren zu berechnen und zu speichern sind. Dies hängt natürlich auch vom verwendeten Lösungsalgorithmus ab.
3. Das Lösen des Gleichungssystems. Wie oben diskutiert, gibt es hier zwei Möglichkeiten, nämlich die komplette Lösung des Gleichungssystems (full matrix – fm) und die iterative Näherung (progressive radiosity – pr).
4. Die Ausgabe der berechneten Energiedichten. Hier stellt sich als wesentliches Problem die Berechnung der Eckfarbwerte aus den in den Schwerpunkten der Flächen gegebenen Energiedichten.

Der Frage nach einer effizienten Lösung des Gleichungssystems gehen wir zuerst nach, da deren Beantwortung Einfluß auf das gesamte Programm hat.

18.1 Lösung des Gleichungssystems

Wie schon gesagt: eine Möglichkeit, das Gleichungssystem zu lösen, besteht darin, die Matrix A zu invertieren und die Lösung anschließend mit $B = A^{-1} \cdot E$ zu berechnen. Die Lösbarkeit kann ohne mathematischen Beweis aufgrund des physikalischen Ansatzes angenommen werden!

Die Invertierung von A kann folgendermaßen berechnet werden [BS81]:

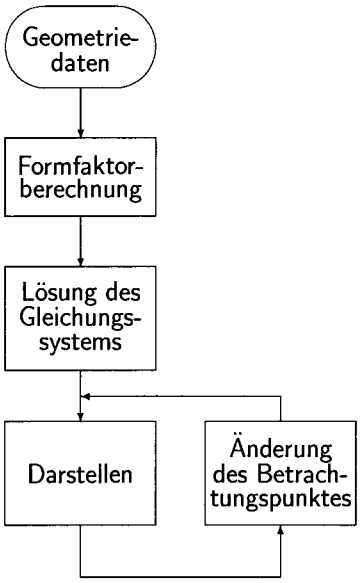


Abb. 18.1 Ablaufplan des Radiosity-Verfahrens

$$A^{-1} = \frac{1}{\det A} \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ A_{21} & A_{22} & \cdots & A_{2n} \\ \vdots & \vdots & & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nn} \end{pmatrix}^T .$$

Die A_{ij} bezeichnen sogenannte adjunkte Matrizen, die wiederum im wesentlichen Unterdeterminanten von A sind. Dieses Verfahren ist sowohl im zeitlichen Verhalten als auch im Speicherplatzverbrauch indiskutabel.

Die zweite Möglichkeit ist, das Gleichungssystem direkt, z.B. mit dem Gauß'schen Algorithmus, zu lösen. Dieser benötigt keinen großen zusätzlichen Speicherplatz, ist aber in der Rechenzeit von der dritten Potenz der Größe der Matrix abhängig. Die Matrix hat eine Dimension entsprechend der Anzahl der Flächen in der Szene. Neben diesem Nachteil können erhebliche numerische Probleme bei der Verwendung dieses Algorithmus auftreten.

Der dritte Weg, das Gleichungssystem zu lösen, ist mittels iterativer Lösungsverfahren. Dazu zählen die sogenannten Gesamtschrittverfahren (Jacobi) und Einzelschrittverfahren (Gauß-Seidel). Die Voraussetzung für die Anwendung beider Verfahren, die Diagonaldominanz der Matrix A, ist für unser Problem erfüllt. Genauere Angaben zu deren Berechnung finden sich z.B. bei Bronstein und Semendjajew [BS81]. Eine Variante des Einzelschrittverfahrens wird im folgenden genauer beschrieben.

18.1.1 Iteratives Lösungsverfahren

Da (Teil-) Ergebnisse eines iterativen Lösungsverfahrens schnell darstellbar sind, den Speicher- und Rechenzeitkapazitäten auf einem PC eher entsprechen und das Verfahren sehr schön in seiner Abfolge zu beobachten ist, haben wir diese Lösung gewählt. Sie entspricht im Prinzip dem zeilenweisen Lösen des Gleichungssystems. Pro Iterationsschritt wird die Fläche gesucht, die (noch) die meiste Energie zu verteilen hat. Deren Energie (`E`) entspricht der Energiedichte der Fläche (Variable `B`) multipliziert mit deren Flächeninhalt. Für diese spezielle Fläche berechnet man alle Formfaktoren zu allen anderen Flächen. Anschließend wird die Energie der Fläche gemäß der Formfaktoren verteilt, die verteilte Energie von der Flächenenergie abgezogen und auf die entsprechenden anderen Flächenenergien addiert. Dieser Iterationsschritt wird wiederholt, so lange nicht alle Energie einer Szene vollständig verteilt ist.

```
REPEAT
  SuchePatchMitGroesstemdBA(..)
  BerechneFormfaktoren(..);
  WandeleFormfaktorenInEnergieUm(..);
  BerechnePunktFarbenAusEnergien(..);
  RendereSzene(..);
UNTIL IstKonvergiert;
```

Das Verfahren konvergiert gegen die fm-Lösung, hat aber den Vorteil, daß das Teilergebnis nach jedem Iterationsschritt dargestellt werden kann (Abb. VIII).

Damit zu Anfang, wenn nur ein geringer Teil der Energie verteilt ist, dennoch ein sinnvolles Bild zustande kommt, wird zusätzlich ein ambienter Term eingeführt (Abb. IX). Dieser Term umfaßt am Anfang die Summe aller Energien der Szene.

$$\sum_{i=1}^{N} E_i$$

Danach entspricht er noch der Summe aller Restenergien, die pro Fläche als Restenergiedichte multipliziert mit der Fläche abgelegt sind.

$$\sum_{i=1}^{N} (\Delta B_i \cdot A_i)$$

Die Gesamtenergie wird für die Darstellung als Energiedichte auf den Gesamtflächeninhalt der Szene bezogen.

$$\frac{\sum_{i=1}^{N} (\Delta B_i \cdot A_i)}{\sum_{i=1}^{N} A_i}$$

Um hieraus den korrekten Term für eine ambiente Beleuchtung zu gewinnen, sind noch mehrfache Reflexionen dieses Lichtes zu berücksichtigen. Dazu bestimmt man die mittlere Reflektivität ρ_{avg} der Szene, die aus den mit der Fläche gewichteten Reflektivitäten der einzelnen Flächen besteht.

$$\rho_{avg} = \frac{\sum_{i=1}^{N}(\rho_i A_i)}{\sum_{i=1}^{N} A_i}$$

Mit Hilfe von ρ_{avg} läßt sich die Gesamtmenge ambienten Lichtes ausdrücken. Diese besteht aus dem oben genannten Term plus der im Mittel reflektierten Energiemenge plus der zwei Mal im Mittel reflektierten Energiemenge, ... usw. Diese unendliche Reihe von Energiemengen drückt sich in einem Faktor R aus.

$$R = 1 + \rho_{avg} + \rho_{avg}^2 + \cdots = \frac{1}{1 - \rho_{avg}}$$

Damit ergibt sich für den ambienten Term endgültig:

$$amb = R \cdot \frac{\sum_{i=1}^{N}(\Delta B_i \cdot A_i)}{\sum_{i=1}^{N} A_i}.$$

Man erkennt, daß die mittlere Reflektivität ρ_{avg} einer Szene kleiner als Eins bleiben muß, sonst konvergiert das Verfahren nicht (und hätte auch keine physikalische Entsprechung, da die alles reflektierende Oberfläche nicht existiert). Haben zum Beispiel zwei Flächen, die sich parallel gegenüber liegen, eine Reflektivität, die nur geringfügig größer als Eins ist, so genügt ein einziges Lichtquantum, um die Energie in's Unendliche steigen zu lassen. Aus diesem Beispiel kann man zwei Dinge lernen:

- Bei der Radiosity-Berechnung nie Reflektivitäten verwenden, die > 1 sind.
- Die Formel für *amb* ist nur eine Näherungsformel, da sie für das genannte Beispiel durchaus einen endlichen Wert liefert.

Noch ein anderes Beispiel macht deutlich, daß der Wert von *amb* mit Vorsicht zu genießen ist:

In einem unendlich schwarzen Raum steht eine unendlich schwarze, geschlossene Kiste, in der sich wiederum viele kleinere Kisten befinden, die aber zur Abwechselung alle weiß sind. Außer der schwarzen Kiste (mit den vielen Unterkisten) sei noch eine Lichtquelle im Raum plaziert. Wie sich jeder leicht vorstellen kann, ist der Raum dunkel, egal wieviele weiße Kisten in der schwarzen sind. Die Formeln für ρ_{avg} bzw. R liefern allerdings einen falschen Wert $\neq 0$.

Die Idee, daß die Formeln dann exakt sind, wenn man nur die Flächen bei der Berechnung berücksichtigt, die irgendwie vom Licht erreicht werden, läßt sich anschaulich leicht dadurch wiederlegen, daß man in die schwarze und alle weißen Kisten ein winziges Loch „bohrt".

Diese Fehler bei der Berechnung von R können zur Folge haben, daß die dargestellte Szene zu hell oder zu dunkel erscheint. Aus diesem Grund ist es sinnvoll, *amb* mit einem Faktor zu gewichten, der vom Benutzer entsprechend eingestellt wird. Wir haben diesen Faktor aufgrund des oben aufgeführten Beispiels „Sichtbarkeitsfaktor" getauft. Das ganze klingt ziemlich unwissenschaftlich und ist es auch. Im Augenblick ist uns allerdings keine bessere Lösung bekannt.

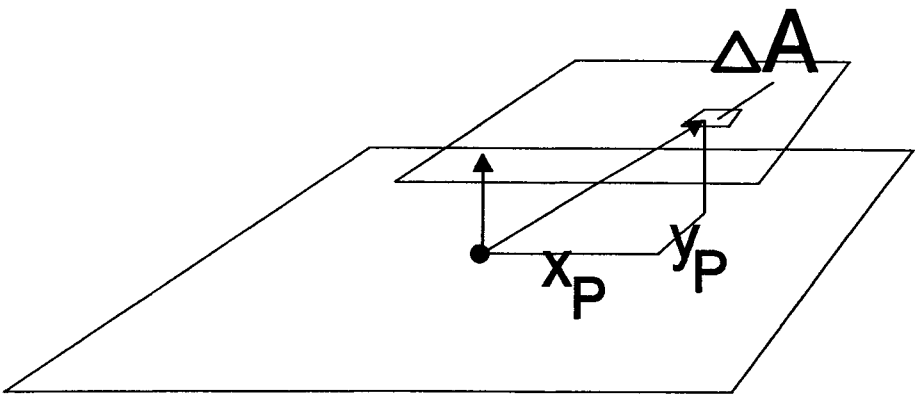


Abb. 18.2 Darstellung der geometrischen Verhältnisse bei der Berechnung der Formfaktoren

18.2 Berechnung der Formfaktoren

Es gibt verschiedene Möglichkeiten, die Formfaktoren zu berechnen. Da die exakte Lösung vom Rechenaufwand her nicht in Frage kommt, verwendet man einen gerasterten Einheitshalbwürfel, oder auch nur dessen „Deckel". Damit erhält man eine einzige Fläche, auf der alle anderen Flächen dargestellt werden müssen, die sogenannte „Singleplane". Um eine höhere Genauigkeit der Berechnung der Formfaktoren zu erreichen, kann das Innere dieser Ebene mit höherer Auflösung gerastert werden als die äußeren Teile. Dies wird durch eine zweite, in Weltkoordinaten kleinere Ebene innerhalb der ersten realisiert. Damit sind, bei gleicher Zahl der Rastereinheiten auf den Ebenen, die Teilformfaktoren auf der inneren Ebene kleiner, es können darauf aber auch kleinere Objekte abgebildet werden.

Die Prozedur `BerechnedFFs` berechnet für beide Ebenen die Teilformfaktoren, die für ein Rasterelement auf der Ebene gültig sind. Die Teilformfaktoren stehen in einem zweidimensionalen Feld, auf das mittels spezieller Routinen zugegriffen werden kann (siehe Kap. 18.4).

Für ein Pixel auf einer Ebene, die mit dem Abstand 1 über einer Fläche lokalisiert ist, berechnet sich der Teilformfaktor aus der Beziehung

$$\Delta F_p = \frac{1}{\pi(x_p^2 + y_p^2 + 1)^2} \cdot \Delta A$$

wenn x_p und y_p die Koordinaten des Pixels sind (Abb. 18.2).

18.3 Struktur des implementierten Verfahrens

Mit allen diesen Informationen versehen, kann bereits die Grobstruktur des Hauptprogramms untersucht werden.

```
LeseGEDDatei(FileName,Szene,..);
DereferenziereSzene(Szene,..);
ErzeugePatchesUndElemente(PatchKantenL,ELKantenL,..);
BerechneFlaechenGroessenUndAmb(..,Amb,GesamtFlaeche,
                               MittlereReflektivitaet);
amb:=amb*VisFak;
BerechnedFFs(..);
REPEAT
  SuchePatchMitGroesstemdBA(Nr)
  BestimmeMatrizenFuerSingleplane(Nr,..)
  ElementeAufSingleplane(..)
  BerechneFormfaktoren(..);
  WandeleFormfaktorenInEnergieUm(Nr,..,VerteilteEnergie);
  amb:=amb-1/(1-MittlereReflektivitaet)*
                        VerteilteEnergie/GesamtFlaeche;
  IF amb<0 THEN amb:=0;
  BerechnePunktFarbenAusEnergien(..);
  RendereSzene(..,amb);
UNTIL IstKonvergiert;
```

`LeseGEDDatei` liest eine Eingabedatei und wandelt sie in die internen Strukturen des Programms um. Mit Hilfe von `DereferenziereSzene` wird die so eingelesene Szene in eine lineare Liste von Flächen und Punkten umgewandelt. Mit `ErzeugePatchesUndElemente` werden die Flächen der erzeugten Liste so unterteilt, daß die Kanten nicht länger als `PatchKantenL` bzw. `ELKantenL` sind.

Bei der Radiosity-Iteration dienen die Patches (das englische Wort für „Flicken") zum Senden der Energie, die Elemente zum Empfangen. Dabei besteht ein Patch immer aus einem oder mehreren Elementen, sozusagen den „Kindern" des Patches. Beim Einsammeln der Energie wird diese in den Elementen gespeichert, die unverteilte Energiedichte wird an das „Mutterpatch" weitergereicht. Beim Verteilen wird die Liste der Flächen nach dem Patch durchsucht, dessen Kinder bis zu diesem Zeitpunkt die meiste Energie gesammelt haben.

Durch die Aufteilung in Patches und Elemente wird einerseits erreicht, daß bei der Darstellung durch die kleinen Elemente relativ hoch aufgelöste Schatten abgebildet werden können. Andererseits ist damit beim Verschießen eine relativ geringe Anzahl von Schritten notwendig. Eine zu grobe Auflösung der Patches ist allerdings auch nicht wünschenswert. Sind diese aus großen Flächenlichtquellen entstanden, verbreiten sie ihre Energie am Anfang der Iterationen. Um zu vermeiden, daß sie durch eine einzige Iteration angenähert werden, sollte bei diesen schon in der Eingabedatei eine Facettierung vorgenommem werden.

`BerechneFlaechenGroessenUndAmb` berechnet für alle – auch neu entstandene – Flächen deren Flächeninhalt und ermittelt den Wert von *amb* wie oben angegeben. Anschließend erfolgt die Multiplikation von `amb` mit dem oben erwähnten Sichtbarkeitsfaktor. In `BerechnedFFs` werden die Teilformfaktoren, die während der Iteration konstant sind, bestimmt.

Schließlich wird die eigentliche Radiosity-Iteration in einer `REPEAT-UNTIL`-Schleife durchgeführt: `SuchePatchMitGroesstemdBA` sucht das Patch mit der

größten zu verteilenden Energie, `BestimmeMatritzenFuerSingleplane` berechnet anhand der Lage dieser Fläche die Matrizen für die Transformation der Elemente auf die Singleplane, `ElementeAufSingleplane` rendert alle Elemente auf die Singleplane und `BerechneFormfaktoren` bestimmt dann mit Hilfe der Deltaformfaktoren aus `BerechnedFFs` die Formfaktoren der Elemente.

Nachdem diese wesentlichen Schritte durchlaufen sind, müssen die Formfaktoren noch mit Hilfe der Reflektivität der Elemente und der verteilten Energie in Energien der Elemente umgerechnet werden, was die Routine `WandeleFormfaktorenInEnergieUm` durchführt. In dieser Routine wird auch die zu verbreitende Energie des Patches, das gerade geschossen hat, zurückgesetzt. Mit

```
amb:=amb-1/(1-MittlereReflektivitaet)*
                VerteilteEnergie/GesamtFlaeche;
IF amb<0 THEN amb:=0;
```

wird der ambiente Term korrigiert. Die Abfrage `IF amb<0` ist nötig, da durch ungeschickte Wahl von `VisFak` vor der Iteration der Wert von `amb` im Laufe der Iteration negativ werden kann.

Bevor dann letztlich die Szene mit `RendereSzene` dargestellt wird, muß noch von den Energien der Flächen auf die Farbe der Punkte geschlossen werden. Dies erledigt die Routine `BerechnePunktFarbenAusEnergien`.

18.3.1 Vor- und Nachteile des iterativen Verfahrens

Experimente mit dem Programm führen zu folgenden Schlußfolgerungen:

1. Die Feinheit der Unterteilung der Singleplane hat großen Einfluß auf die Qualität der erzeugten Bilder. Dies hängt im wesentlichen mit Abtastfehlern zusammen, die sich unweigerlich einstellen, wenn eine zu geringe Auflösung gewählt wird. Ein Ausweg hieraus ist die exakte Projektion der Flächen der Szene auf die Ebene und die Aufhebung der Rasterisierung der Ebene.
   Das Verfahren läuft dann so ab, daß zuerst alle Eckpunkte einer Fläche auf die – nun unendlich ausgedehnte – Ebene projiziert werden. Man erhält dann also die projizierte Fläche. Diese Fläche wird mit allen anderen Flächen verglichen, um gegenseitige Verdeckungen zu beseitigen. Dabei entstehen modifizierte projizierte Flächen, im allgemeinen Polygone. Da eine exakte Berechnung des Formfaktors eines Dreieckes bekannt ist, kann nach einer Zerlegung des Polygons in Dreiecke der Formfaktor für die Fläche mit Berücksichtigung der Verdeckungen exakt berechnet werden. Allerdings ist diese Vorgehensweise auch wieder eine Stufe aufwendiger als die vorher beschriebene.
   Selbst wenn diese Lösung unter Aufwandsaspekten akzeptiert wird, verschärft sich das Problem mit der Aufhebung der Beschränkung auf planare Flächen. Was also soll man tun, wenn die eingegebene Szene auf einmal B-Spline- oder Beziér-Flächen enthält?

2. Eine weitere Fehlerquelle ist die Endlichkeit der Singleplane. Dadurch werden nicht alle möglichen anderen Flächen der Szene mit Einfluß auf die ausgewählte erfaßt und die Summe aller Teilformfaktoren bleibt geringer als Eins. Dem kann dadurch Abhilfe geschaffen werden, daß alle paar Iterationsschritte ein Halbwürfel statt der Ebene verwendet wird.
3. Eine adaptive Szenenunterteilung, also die Unterteilung von Nachbarflächen bei zu großer Energiedichtedifferenz und damit Verletzung der Homogenitätsforderung, bringt mehrere Probleme mit sich. Der visuell am stärksten beobachtbare Effekt tritt aber dadurch auf, daß die korrekt unterteilten Flächen so klein werden, daß sie beim Empfangen von Energie dann nicht mehr zum Energieaustausch beitragen. Dieser Effekt wird dann als „Streuselkuchen"-Effekt sichtbar und hat einen der Autor(inn)en bis kurz vor den Wahnsinn getrieben.
   Der Effekt kann nur durch die bereits beschriebene Methode der Formfaktorberechnung behoben werden.

Viele Probleme im Zusammenhang mit dem Radiosity-Verfahren sind noch nicht gelöst oder Gegenstand der Forschung. Neben der Beschäftigung mit den Problemen, die durch die Näherung der Formfaktoren entstehen, ist ein zweiter großer Bereich die Untersuchung der Vereinbarkeit von diffuser und spiegelnder Reflexion.

18.4 Die Unit PRIMRAD, Radiosity zum Anfassen

Die Unit PRIMRAD stellt die wesentlichen Routinen für die Implementierung des Radiosity-Verfahrens in Pascal zur Verfügung. Im Interfaceteil dieser Unit sind folgende Prozeduren definiert:

```
PROCEDURE ErzeugePatchesUndElemente(..)
PROCEDURE BerechneFlaechenGroessenUndAmb(..)
PROCEDURE BerechnedFFs(..)
PROCEDURE SuchePatchMitGroesstemdBA(..)
PROCEDURE BestimmeMatrizenFuerSingleplane(..)
PROCEDURE ElementeAufSingleplane(..)
PROCEDURE BerechneFormfaktoren(..)
PROCEDURE WandeleFormfaktorenInEnergieUm(..)
PROCEDURE MittlereBeteiligteFlaechen(..)
PROCEDURE BerechnePunktFarbenAusEnergien(..)
```

PRIMRAD benutzt wie alle anderen Units PRIMGLOB für die Parameterlisten im Schnittstellenteil und die Units PRIMVIEW, PRIMMEM und PRIMOUT.

PRIMVIEW wird für die Routine BestimmeMatrizenFuerSingleplane benutzt, PRIMMEM dient zum Ansprechen des XMS-Speichers und PRIMOUT wird lediglich im Fehlerfall benötigt (PROCEDURE Fehler; die einzelnen Units werden in den Kap. 9.2, C und C.3 beschrieben).

Im folgenden werden alle Routinen, die im Interfaceteil von PRIMRAD definiert sind, relativ ausführlich erläutert. Damit soll ihrer Wichtigkeit für das gesamte Programm PRIM Rechnung getragen werden.

18.4.1 Die Prozedur ErzeugePatchesUndElemente

```
PROCEDURE ErzeugePatchesUndElemente(
            maxPatchL, maxElementL: SINGLE;
            VAR AnzahlFlaechen, AnzahlPunkte,
                AnzahlPatches: LONGINT);
```

Als Eingangsparameter der Prozedur ErzeugePatchesUndElemente dienen die maximalen Längen maxPatchL und maxElementL, welche die Kanten der *Patches* bzw. *Elemente* nach der Unterteilung haben dürfen. Dabei gelten die Längenangaben im Weltkoordinatensystem. In Konkurrenz zu maxPatchL steht ein Wert MaximaleKantenLänge, der für jede Fläche definiert ist, also in der Flächenliste selbst gespeichert ist. Dieser Wert wird anhand der Facettierung der Fläche bestimmt. Der kleinere der beiden Werte wird als Unterteilungskriterium verwendet. Auf diese Art dient ErzeugePatchesUndElemente auch zur Facettierung der ursprünglichen Szene.

Die Übergabeparameter AnzahlFlaechen und AnzahlPunkte müssen beim Aufruf der Routine die entsprechenden aktuellen Werte haben. Hierfür können die Ergebnisse der Routinen DereferenziereSzene benutzt werden. Da ErzeugePatchesUndElemente neue Flächen und Punkte generiert, verändern sich auch AnzahlFlaechen und AnzahlPunkte. Die Routine gibt diese veränderten Werte an das aufrufende Programm zurück.

AnzahlPatches gibt dem aufrufenden Programm an, wieviele Patches die zu berechnende Szene enthält. Diese Patches stehen genauso wie die erzeugten Elemente in der globalen Flächenliste, wobei die Patches immer am Anfang der Liste stehen.

Nachdem nun die Übergabeparameter erklärt sind, einige Worte zur Art und Weise, wie die Routine arbeitet: Zuerst werden in einer WHILE-Schleife aus den ursprünglichen Flächen Patches erzeugt. Diese WHILE-Schleife wird solange durchlaufen, bis keine Unterteilungen mehr stattfinden.

Algorithmus 18.1 (Umwandlung von Flächen zu Patches)

WurdeGeteilt := TRUE.
SOLANGE WurdeGeteilt = TRUE.
WurdeGeteilt := FALSE.
FÜR alle Flächen der Flächenliste:
WENN irgendeine Kante der Fläche länger als ..
.. maxPatchL oder Flaeche.MaximaleKantenlaenge, DANN:
Unterteile Fläche, wobei die zu unterteilende Fläche durch ..
.. eine ihrer Kinder ersetzt wird. Alle Kinder sind Patch ..
.. und Element.
WurdeGeteilt:=TRUE.
ENDE WENN.
ENDE FÜR alle Flächen der Flächenliste.
ENDE SOLANGE WurdeGeteilt = TRUE.

Nachdem durch diese Unterteilung die gesamte Flächenliste aus Patches besteht, werden die Patches mit einem sehr ähnlichen Verfahren in Elemente unterteilt. Im Gegensatz zum gerade vorgestellten Verfahren ersetzen die Elemente die Patches nicht, sondern sie erhalten einen Index auf das Patch, durch das sie mittels Unterteilung erzeugt werden. Über diesen Index kann bei der Radiosity-Iteration beim Einsammeln der von den Elementen empfangenen Energien diese an das entsprechende Patch zum zukünftigen Verteilen weitergereicht werden.

Algorithmus 18.2 (Umwandlung von Patches in Elemente)

WurdeGeteilt := FALSE.
FÜR alle Flächen(=Patches) der Flächenliste:
WENN irgendeine Kante der Fläche länger als maxElementL, DANN:
Unterteile Fläche, wobei der Index der zu unterteilenden ..
.. Fläche in den neuen Kindern gespeichert wird.
Zu Unterteilende Fläche.IstElement := FALSE.
Kinderflächen.IstPatch := FALSE.
WurdeGeteilt := TRUE.
ENDE WENN.
ENDE FÜR alle Flächen der Flächenliste.
SOLANGE WurdeGeteilt = TRUE:
FÜR alle Flächen der Flächenliste, für die Fläche.IstElement = TRUE:
WENN irgendeine Kante der Fläche länger als maxElementL, DANN:
Unterteile Fläche, wobei die zu unterteilende Fläche ..
.. durch eine ihrer Kinder ersetzt wird. Alle Kinder ..
.. sind nicht Patch, aber Element.
Setze den PatchIndex der Kinder auf den PatchIndex der Mutter.
WurdeGeteilt := TRUE.
ENDE WENN.
ENDE FÜR alle Flächen der Flächenliste.
ENDE SOLANGE WurdeGeteilt = TRUE.

18.4.2 Nachbarschaften

Bei der Radiosity-Iteration wird die empfangene Energie der Elemente bestimmt. Bei der Darstellung der Elemente benötigt man allerdings nicht die Energie der Elemente, sondern die Farbe der Eckpunkte. Für die Umrechnung der Elementenergie in die Punktfarbe ist also ein besonderer Algorithmus notwendig. In unserer Implementierung wird dieses Problem mittels Nachbarschaftsbeziehungen zwischen nebeneinanderliegenden Flächen gelöst.

Als Farbe eines Punktes gilt der Mittelwert aller empfangenen Energien aller Flächen, von denen der Punkt Eckpunkt ist, multipliziert mit der Reflektivität der Fläche, für die der Punkt als Eckpunkt dargestellt werden soll.

$$C_{P_{ij}} = \rho_j \cdot \frac{\sum_{N_{ij}} B_{N_{ij}} \cdot A_{N_{ij}}}{\sum_{N_{ij}} A_{N_{ij}}},$$

wobei P_{ij} der i-te Eckpunkt der Fläche j ist und N_{ij} die Nachbarn der Fläche j an der Ecke i bezeichnet.

Um nun die Nachbarn N_{ij} bestimmen zu können, wird pro Kante einer Fläche der Index der Flächen gespeichert, die ebenfalls an dieser Kante hängen. Hierzu dient das Feld `NachbarIndizes` von `GlobaleFlaecheTyp`. Die Nach-

barschaften werden beim Einlesen einer GED-Datei für polygonale Teilobjekte automatisch nach folgendem Algorithmus erzeugt:

Algorithmus 18.3 (Bestimme Nachbarn von Polygon-Teilobjekten)

(j ist der Index der Flächen, für die die Nachbarn bestimmt ..*
*.. werden sollen *)*
FÜR alle Flächen F_j des Teilobjektes:
FÜR jede Kante i der Fläche F_j:
Setze ein Flag $Flag_i$:= FALSE.
Setze $NachbarIndex_i$ auf j.
($Flag_i$ gibt an, ob für Kante i schon eine Nachbarfläche ..*
.. gefunden wurde, $NachbarIndex_i$ ist der Index dieser ..
.. Nachbarfläche. Hat die Fläche für die Kante i keinen ..
.. Nachbarn, so wird als NachbarIndex ihre eigenen Nummer ..
*.. gespeichert. *)*
ENDE FÜR jede Kante i.
k = 0. (k ist Index der Fläche, für die untersucht werden soll, ..*
*.. ob sie Nachbar von F_j ist *)*
WIEDERHOLE:
WENN $k \neq j$, DANN:
FÜR alle Kanten K_j der Fläche k:
FÜR alle Kanten K_i der Fläche j:
WENN erster Punkt von K_j = zweiter Punkt von K_i und ..
.. zweiter Punkt von K_j = erster Punkt von K_i, DANN:
WENN $Flag_i$ = TRUE, DANN:
Warnung ausgeben: Mehr als zwei Flächen an einer Kante.
SONST $Flag_i$:= TRUE und $NachbarIndex_i$:= k.
ENDE WENN.
ENDE FÜR alle Kanten K_i.
ENDE WENN $k \neq j$, DANN. ENDE FÜR alle Kanten K_j.
BIS alle $Flag_i$ = TRUE oder alle Flächen mit F_j verglichen wurden.
ENDE FÜR alle Flächen F_j.

In Abb. 18.3 ist dazu ein Beispiel dargestellt. Die Fläche 10 habe die Eckpunkte mit den Nummern 12, 17, 30 und 22. Die Fläche 15 habe die Eckpunkte 30, 17 und 33.

Der obige Algorithmus würde dann als Nachbarindex der Kante 1 (von Punkt 17 nach Punkt 30) von Fläche 10 die Fläche 15 und als Nachbarindex der Kante 0 (von Punkt 30 nach Punkt 17) von Fläche 15 die Fläche 10 bestimmen, wenn die Indizierung der Kanten und Nachbarindizes bei 0 beginnt.

Der Grund für das „kreuzweise" Vergleichen der Punkte ist aus Abb. 18.3 leicht einzusehen: Flächen mit gleichem Umlaufsinn haben für gemeinsame Kanten immer verschiedene Richtungen. Der Aufwand des Algorithmus ist $O(N^2)$ mit N = Anzahl der Flächen. Hier kann sicher eine Menge optimiert werden, was wir den Lesern gern überlassen.

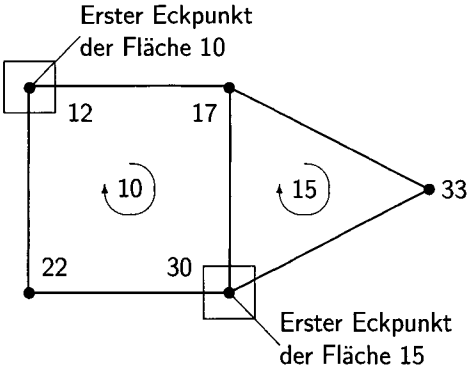


Abb. 18.3 Beispiel zur Berechnung der Nachbarindizes einer Fläche

Für vordefinierte Körper, die durch Flächen angenähert werden, sind die Nachbarschaftsbeziehungen a priori gegeben. Sie müssen also nicht mit Algorithmus 13.6 bestimmt werden.

Zusammenfassend läßt sich über die Nachbarschaften von Flächen sagen:

1. Für alle Kanten der Fläche ist ein Nachbarindex definiert. Existiert an einer Kante kein Nachbar, so ist der Nachbarindex auf den Index der Fläche selbst gesetzt.
2. Zu einer Kante i gibt es genau einen Punkt P_i, der als Repräsentant der Kante verwendet werden kann. In Abb. 18.3 ist z.B. Punkt 17 Repräsentant der Kante 1 von Fläche 10 und Repräsentant der Kante 1 von Fläche 15.
3. Jede Fläche hat genausoviele Kanten wie Punkte.

Bei der Aufteilung in Patches und Elemente müssen die vorhandenen Nachbarschaftsbeziehungen erhalten bleiben. Dies führt zu Komplikationen, wenn eine von zwei benachbarten Flächen i und j unterteilt wird, die andere aber nicht. In diesem Fall spricht man von einem T-Stück (Abb. 18.4).

Im Normalfall kann dies nicht passieren, da anhand der Kantenlänge entschieden wird, ob eine Fläche unterteilt werden soll oder nicht. Wenn für die Fläche i die Kante unterteilt werden soll, so wird auch bei der Fläche j eine Unterteilung stattfinden, wenn i und j diese Kante gemeinsam haben.

Daß T-Stücke trotzdem auftreten können, liegt an der Art, wie Polygone unterteilt werden. Nach dem Einlesen und Dereferenzieren von GED-Dateien besteht die Flächenliste ausschließlich aus Drei- und Vierecken. N-Ecke werden entsprechend unterteilt. Für Dreiecke kann es vorkommen, daß eine, zwei oder alle drei Kanten geteilt werden müssen, bei Vierecken sind es eine bis vier Kanten (Abb. 18.5).

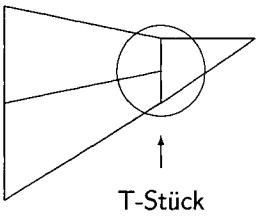


Abb. 18.4 Beispiel für ein T-Stück

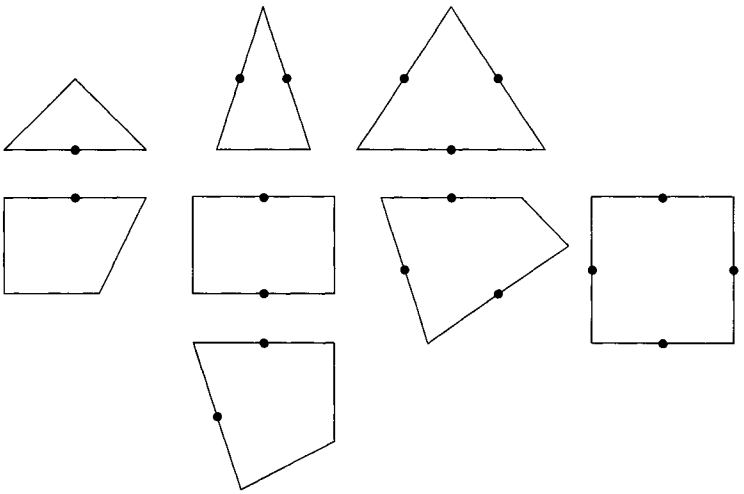


Abb. 18.5 Unterteilung von Drei- und Vierecken

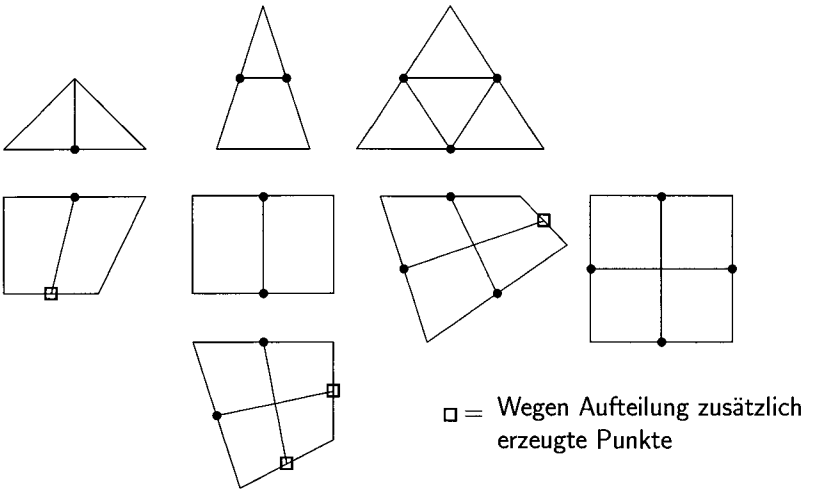


Abb. 18.6 Aus den Unterteilungen erzeugte Flächen

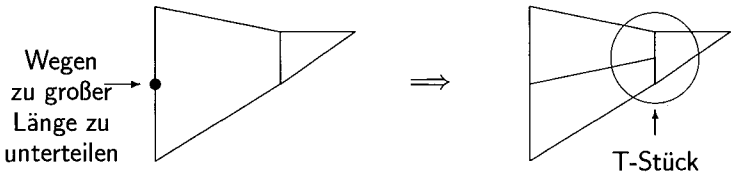
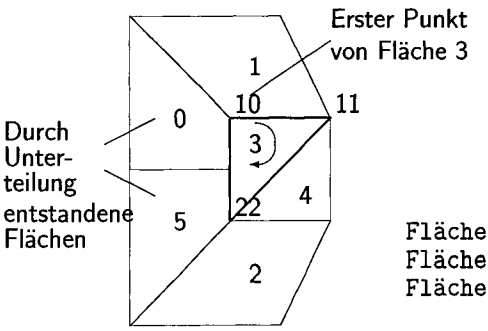


Abb. 18.7 Durch Unterteilung entstandenes T-Stück

Aus diesen Flächen werden durch Unterteilung die in Abb. 18.6 dargestellten Flächen erzeugt. Man sieht, daß bei den Vierecken unter Umständen an Kanten, die normalerweise nicht unterteilt würden, neue Kantenpunkte eingetragen werden. Diese Kanten sind Kandidaten für T-Stücke. So hat z.B. das Dreieck in Abb. 18.7 an einer Kante gleich zwei Nachbarn. Um auch diesen Fall handhaben zu können, ist zusätzlich in jeder Fläche ein Feld `Nachbar1Indizes` definiert, in welcher der Index der eventuell zweiten Nachbarfläche einer Kante gespeichert ist. Ist die Kante kein T-Stück, so wird der entsprechende Index auf -1 gesetzt.

Die Abb. 18.8 zeigt die Werte von `NachbarIndizes`, `Nachbar1Indizes` und `Punktindizes` für das mittlere Dreieck exemplarisch. Aus der Abbildung ist zu erkennen, daß Fläche 5 als Nachbar und Fläche 0 als `Nachbar1` der dritten Kante von Fläche 3 gespeichert ist.

Wer nun denkt, daß hiermit das Thema Nachbarschaften abgeschlossen ist, täuscht sich. Was passiert, wenn in Abb. 18.8 nach dem Unterteilen die linken




```
Fläche3.PunktIndizes: 10, 11, 22
Fläche3.NachbarIndizes: 1, 4, 5
Fläche3.Nachbar1Indizes: -1, -1, 0
```

Abb. 18.8 Beispiel für daie Werte von `NachbarIndizes`, `Nachbar1Indizes` und `Punktindizes`

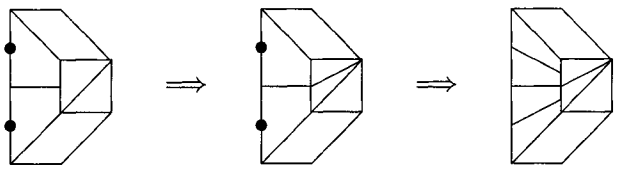


Abb. 18.9 Unterteilung einer Fläche trotz genügend kurzer Kanten

Kanten von Fläche 0 oder 5 immer noch zu lang sind? In den Algorithmen 18.1 und 18.2 kann dieses ab dem zweiten Durchlauf durchaus passieren. Nun, in diesem Fall wird vor der Unterteilung Fläche 3 unterteilt, obwohl keine Ihrer Kanten zu lang ist (Abb. 18.9).

Die gerade vorgestellten Verfahren sind in den Routinen `Unterteile-Dreieck`, und `UnterteileViereck` implementiert, die lokal innerhalb der Unit `PRIMRAD` definiert sind. Diese Routinen haben übrigens einen nicht unwesentlichen Teil der Zeit in Anspruch genommen, die wir für die Implementierung des Radiosity-Verfahrens benötigt haben.

18.4.3 Die Prozedur `BerechneFlaechenGroessenUndAmb`

```
PROCEDURE BerechneFlaechenGroessenUndAmb(AnzahlFlaechen: LONGINT;
            VAR Amb                    : FarbVektorTYP;
            VAR GesamtFlaeche          : SINGLE;
            VAR MittlereReflektivitaet : FarbVektorTYP);
```

In dieser Prozedur wird für alle Flächen der Flächenliste der Flächeninhalt bestimmt. Außerdem erfolgt die Berechnung des Anfangswertes für den ambienten Anteil. `Amb` und die `GesamtFlaeche` (Summe aller Flächeninhalte) sowie die

`MittlereReflektivitaet`, die ρ_{avg} entspricht, gibt die Prozedur an das aufrufende Programm zurück. Die Flächeninhalte der einzelnen Flächen werden im Eintrag `FlaechenInhalt` gespeichert.

Zur Erinnerung: Sind P_1, P_2 und P_3 die Punkte eines Dreiecks, so ergibt sich der Flächeninhalt des Dreiecks zu

$$F_\triangle = \frac{1}{2}(P_2 - P_1) \times (P_3 - P_1).$$

Der Flächeninhalt eines konvexen Vierecks ist entsprechend

$$F_\square = \frac{1}{2}(P_2 - P_1) \times (P_3 - P_1) + \frac{1}{2}(P_3 - P_1) \times (P_4 - P_1).$$

18.4.4 Die Prozedur `BerechnedFFs`

```
PROCEDURE BerechnedFFs(PlaneSize1WC: SINGLE;
                       PlaneSize2WC: SINGLE;
                       PlaneSizeDC : INTEGER);
```

`BerechnedFFs` bestimmt für eine gegebene Auflösung der Singleplanes die Deltaformfaktoren. Diese bleiben während der Radiosity-Iteration konstant und müssen deshalb nur einmal berechnet werden. In der Routine werden die Deltaformfaktoren für die grobe und für die feine Singleplane bestimmt, so daß die Breite dieser beiden Ebenen im Weltkoordinatensystem übergeben werden muß. `PlaneSizeDC` gibt die Auflösung der Singleplanes in Pixeln an. Sie ist in unserer Implementierung für die grobe und feine Singleplane identisch.

Vor dem Ablauf von `BerechnedFFs` muß unbedingt die Routine `AllokiereIBuffer` der Unit `PRIMMEM` aufgerufen werden. Diese Routine stellt nicht nur den Speicher für den Item-Buffer der Singleplanes, sondern auch den Speicher für die Deltaformfaktoren zur Verfügung.

`BerechnedFFs` bestimmt die Deltaformfaktoren nur für eine halbe Singleplane, um Speicherplatz zu sparen. Theoretisch reicht es wegen der Symmetrie der Deltaformfaktoren sogar aus, nur den ersten Oktanten zu speichern, es ist dann allerdings relativ mühsam, auf die Elemente dieses „$1\frac{1}{2}$dimensionalen" Feldes zuzugreifen [CP91]. Um zu vermeiden, daß dort, wo die feine Singleplane zur Wirkung kommt, die Formfaktoren von grober und feiner Ebene sich nicht überlagern, werden in der groben Ebene an dieser Stelle Nullen gespeichert. Die grobe Singleplane hat also in der Mitte ein „Loch".

18.4.5 Die Prozedur `SuchePatchMitGroesstemdBA`

```
PROCEDURE SuchePatchMitGroesstemdBA(AnzahlPatches: LONGINT;
                                    VAR PatchNr  : LONGINT);
```

Die Routine bestimmt während der Radiosity-Iteration das Patch, das die größte unverteilte Energie hat. Hierzu wird die Anzahl der Patches übergeben, als Er-

gebnis wird die Nummer des gefundenen Patches zurückgegeben. Zur Erinnerung: Die Patches stehen am Anfang der Flächenliste, so daß es ausreicht, nur die ersten `AnzahlPatches` Flächen zu durchsuchen.

18.4.6 Die Prozedur `BestimmeMatrizenFuerSingleplane`

```
PROCEDURE BestimmeMatrizenFuerSingleplane(
            FlaechenNr              : LONGINT;
            Groesse1DerSingleplaneWC: SINGLE;
            Groesse2DerSingleplaneWC: SINGLE;
            GroesseDerSingleplaneDC : INTEGER;
            VAR WC_NPC1, WC_NPC2,
                NPC_WC1, NPC_WC2,
                NPC_DC              : Mat4TYP);
```

Das Rendern der Szene auf die beiden Singleplanes unterscheidet sich vom „normalen" Rendern lediglich dadurch, daß nicht Farben, sondern Flächenindizes „dargestellt" werden. Das bedeutet aber auch, daß die gleichen Routinen zur Bestimmung der Transformationsmatrizen wie beim normalen Rendern herangezogen werden können.

Hierbei gilt folgende Zuordnung:

Auge	⟵	Mittelpunkt der Fläche `FlächenNr`
ReferenzPunkt	⟵	Mittelpunkt der Fläche + Normale der Fläche. Die Normale der Fläche hat die Länge 1, so daß der Referenzpunkt im Abstand 1 vor dem Auge liegt.
UP-Vektor	⟵	beliebig (Zufallsvektor)
Größe der Projektionsebene in der Welt	⟵	Größe der Singleplanes 1 und 2 in der Welt

Anhand dieser Angaben bestimmt `SetzeBetrachter` innerhalb von `BestimmeMatrizenFuerSingleplane` die Transformationsmatrizen `WC_NPC1` und `NPC_WC1` für die erste bzw. `WC_NPC2` und `NPC_WC2` für die zweite Singleplane. Die Matrizen werden dann beim Rendern auf die Singleplane für die Projektion (WC_NPC) und für das Backface-Culling (NPC_WC) verwendet.

Genauso wie beim normalen Rendern in den z-Buffer wird auch beim Rendern auf die Singleplane eine vordere und eine hintere Clipebene benötigt. Der Abstand zur Projektionsebene ist für die vordere Clipebene $1 - \epsilon$. Hierbei ist ϵ ein kleiner Wert (in der aktuellen Implementierung 0.05), der verhindert, daß die vordere Clipebene „durch's Auge" geht, was zu undefinierten Matrizen führte. Ein zu kleiner Wert für ϵ hätte zur Folge, daß für Punkte, die etwas weiter von der zu verteilenden Fläche entfernt sind, die diskrete, aber in der Tiefe nicht lineare Auflösung des z-Buffers sehr gering wäre.

Der Abstand für die hintere Clipebene hängt von der Größe der darzustellenden Szene ab. Er sollte kleiner als der Wert $-(d_{max} - 1)$ sein, um sicher zu

stellen, daß beim Verteilen von Energie nicht „in's Leere“ geschossen wird. d_{max} ist der maximale Abstand zweier Punkte der Szene. Er ist in unserer Implementierung konstant auf 6.0 gesetzt.

Die Matrix NPC_DC, die wegen der gleichen Pixelauflösung von grober und feiner Singleplane für beide identisch ist, wird mittels SetzeBildschirm bestimmt, wobei folgende Zuordnung gilt:

x-Wert linke obere Ecke	⟵	0
y-Wert linke obere Ecke	⟵	0
Breite	⟵	GroesseDerSingleplaneDC
Höhe	⟵	GroesseDerSingleplaneDC

18.4.7 Die Prozedur ElementeAufSingleplane

```
PROCEDURE ElementeAufSingleplane(
            NrDesVerteilendenPatches    : LONGINT;
            WC_NPC, NPC_WC, NPC_DC      : Mat4TYP;
            AnzahlFlaechen, AnzahlPunkte: LONGINT;
            ZeigePlane                  : BOOLEAN);
```

Diese Routine erledigt das Rendern der Elemente auf die Singleplane. Genauso, wie es für die Routine BerechnedFFs nötig ist, den entsprechenden Speicher vor dem Aufruf anzufordern, ist es hier erforderlich, vor der Iteration den benötigten Speicher für den z-Buffer der Singleplane mit Hilfe der Routine AllokiereZBuffer zu reservieren.

Als erstes wird innerhalb der Routine der gesamte z-Buffer sowie der gesamte Item-Buffer gelöscht. Für den Item-Buffer bedeutet dies, daß alle „Pixel“ den Wert -1 erhalten. Beim Bestimmen der Formfaktoren zeigt eine -1 im Item-Buffer an, daß auf das entsprechende Pixel kein Element abgebildet wurde. Nach dem Löschen der beiden Buffer läuft in etwa folgender Algorithmus ab:

Algorithmus 18.4 (Elemente auf Singleplane darstellen)

FÜR alle Punkte:
Transformiere die Punkte mit Hilfe von WC_NPC in's NPC.
Bestimme für die NPC-Koordinaten der Punkte den Clipcode.
WENN Clipcode = Innerhalb, DANN transformiere Punkt in's DC.
ENDE FÜR alle Punkte.
FÜR alle Flächen:
WENN Fläche Element ist und der Patchindex ..
.. der Fläche ≠ NrDesVerteilendenPatches und die Fläche ..
.. keine Rückfläche ist, DANN:
WENN alle Clipcodes der Eckpunkte dieser Fläche = Innerhalb, DANN:
Rendere Fläche auf Singleplane.
SONST, WENN Fläche nicht komplett außerhalb, DANN:
Clippe Fläche.
Rendere geclippte Fläche auf Singleplane.
ENDE WENN, SONST.
ENDE WENN Fläche Element und ...
ENDE FÜR alle Flächen.

Die Matrizen `WC_NPC` und `NPC_DC` werden für die Projektion auf die Singleplane, die Matrix `NPC_WC` für den Rückflächentest benötigt. Ist `ZeigePlane = TRUE`, so wird nach dem Rendern auf die Singleplane der Item-Buffer angezeigt.

`ElementeAufSingleplane` rendert die Szene nur einmal, d.h. die Routine muß zweimal aufgerufen werden: Einmal für die grobe, einmal für die feine Singleplane, jeweils mit den entsprechenden Matrizen, die mit Hilfe von `BestimmeMatrizenFuerSingleplane` berechnet wurden.

18.4.8 Die Prozedur `BerechneFormfaktoren`

```
PROCEDURE BerechneFormfaktoren(GroesseDerSingleplane: INTEGER;
                               AnzahlFlaechen       : LONGINT;
                               IstErstePlane        : BOOLEAN);
```

Aufgabe von `BerechneFormfaktoren` ist es, aus den Werten im Item-Buffer und den entsprechenden Deltaformfaktoren die Formfaktoren der Elemente zu bestimmen. Zu diesem Zweck wird wie folgt vorgegangen.

Algorithmus 18.5 (Berechne Formfaktoren)

FÜR $y = 0$ *bis GroesseDerSingleplane*-1*:*
  FÜR $x = 0$ *bis GroesseDerSingleplane*-1*:*
    I = *Item-Buffer*$[x, y]$.
    WENN $I \neq -1$, *DANN: (* ≠ Hintergrund *)*
      WENN IstErstePlane, DANN:
        dFF = Deltaformfaktor$[x, y]$ *der groben Plane.*
      SONST:
        dFF = Deltaformfaktor$[x, y]$ *der feinen Plane.*
      ENDE WENN, SONST.
      Addiere dFF zur Fläche I.
    ENDE WENN $I \neq -1$.
  ENDE FÜR $x = 0$ *bis GroesseDerSingleplane*-1.
ENDE FÜR $y = 0$ *bis GroesseDerSingleplane*-1.

Dieser Algorithmus entspricht also genau der dargestellten Vorgehensweise bei der Berechnung des Radiosity-Algorithmus. Je nach Wert von `IstErstePlane` werden die Deltaformfaktoren der ersten bzw. der zweiten Singleplane berücksichtigt. Der aufgeführte Algorithmus setzt voraus, daß die Teilformfaktoren zu Beginn auf 0 initialisiert sind, was vor der ersten Iteration durch `ReadGED` bzw. `DerefGED` und ab der ersten Iteration durch `WandeleFormfaktorenInEnergieUm` erledigt wird.

18.4.9 Die Prozedur `WandeleFormfaktorenInEnergieUm`

```
PROCEDURE WandeleFormfaktorenInEnergieUm(
            NrDesVerteilendenPatches: LONGINT;
            AnzahlFlaechen          : LONGINT;
            VAR VerteilteEnergie    : FarbVektorTYP);
```

Dank der diversen Routinen der Unit `PRIMRAD` ist es für `WandeleFormfaktorenInEnergieUm` relativ einfach, die Aufgabe zu erledigen, die der Prozedur ihren Namen gab. Im einzelnen ist dies:

1. Anhand der zu verteilenden Energie `dBA` des *verteilenden* Patches und der Formfaktoren der *empfangenden* Elemente wird die empfangene Energie der Elemente korrigiert. Mit anderen Worten: die neu empfangene Energie wird zu der aus vorangehenden Iterationsschritten aufaddiert.
2. Die von den Elementen reflektierte Energie, also die empfangene Energie, multipliziert mit der Reflektivität der Elemente, wird an das „Mutter"-Patch weitergeleitet und dort aufaddiert.
3. Am Ende wird `dBA` des verteilenden Patches auf 0 (schwarz) gesetzt.
4. Der Wert für die verteilte Energie, der zur Korrektur des ambienten Anteils verwendet wird, ergibt sich aus der Differenz der Energie, die verteilt werden soll und der Energie `dBA`, die nach dieser Iteration nach wie vor „unsichtbar" auf den Patches sitzt, die diese Energie eingefangen haben.

Mathematischer ausgedrückt:

$$\begin{aligned}
\mathcal{R}_{ji} &= dBA_i \cdot F_{ji} \\
B_i &\leftarrow B_i + \frac{\mathcal{R}_{ji}}{A_i} \\
\mathcal{R}'_{ji} &= \rho_i \cdot \mathcal{R}_{ji} \\
dBA_{P_i} &\leftarrow dBA_{P_i} + \mathcal{R}'_{ji} \\
BA_{vj} &= dBA_j - \sum_i \mathcal{R}'_{ji} \\
dBA_j &\leftarrow 0
\end{aligned}$$

wobei die Terme der Tabelle 18.1 verwendet wurden.

Tabelle 18.1 Terme, die zur Berechnung der Formfaktoren in Energien benutzt werden

Bezeichnung	Bedeutung
$\mathcal{R}_{ji}$	die vom Element i empfangene Radiosity des Patches j
dBA_j	von Patch j zu verschießende Energie
F_{ji}	Formfaktor von Patch j nach Element i
B_i	empfangene Energie des Elementes i
A_i	Flächeninhalt des Elementes i
$\mathcal{R}'_{ji}$	vom Element i zurückgeworfene Radiosity des Patches j
ρ_i	Reflektivität des Elementes i
dBA_{P_i}	vom Patch, das „Mutter" des Elementes i ist, zu verschießende Energie
BA_{vj}	verteilte Energie beim „Schuß" von Patch j

Für den Programmierer bzw. die Programmiererin:

Algorithmus 18.6 (Wandele Formfaktoren in Energien um)

VerteilteEnergie := *VerteilendesPatch.dBA.*
FÜR alle Flächen, die Element sind:
  R := *VerteilendesPatch.dBA* · *Flaeche.Formfaktor.*
  Flaeche.B := *Fläche.B* +*R*/*Flaeche.A.*
  R' := *R*· *Flaeche.Reflektivitaet.*
  Addiere R' zu dBA des Patches mit der Nummer Flaeche.PatchIndex.
  VerteilteEnergie := *VerteilteEnergie*–*R'.*
ENDE FÜR alle Flächen.
VerteilendesPatch.dBA := *0.*

18.4.10 Die Prozeduren MittlereBeteiligteFlaechen und BerechnePunktFarbenAusEnergien

```
PROCEDURE MittlereBeteiligteFlaechen(aNr: LONGINT;
            vtxNr          : INTEGER;
            DieseFlaeche   : GlobaleFlaecheTYP;
            VAR DieserPunkt: PunktTYP);
PROCEDURE BerechnePunktFarbenAusEnergien(AnzahlFlaechen: LONGINT);
```

MittlereBeteiligteFlaechen und BerechnePunktFarbenAusEnergien gehören streng genommen nicht mehr zur Radiosity-Iteration. Der Aufruf dieser Routinen ist nur nötig, wenn das Ergebnis einer Iteration angezeigt werden soll. Dank der bekannten Nachbarschaftsbeziehungen der Flächen ist es möglich, mit recht geringem Aufwand von den diversen Flächenenergien, die einen Punkt umgeben, auf dessen Farbe zu schließen.

Zu diesem Zweck werden für jeden Punkt einer Fläche die empfangenen Energien $B \cdot A$ aller diesen Punkt umgebenden Flächen aufaddiert. Diese Energiesumme, geteilt durch die Summe aller Flächeninhalte der beteiligten Flächen, ergibt dann die empfangene Energiedichte an diesem Punkt.

$$C_{P_i} = \frac{\sum_{N_i} B_{N_i} \cdot A_{N_i}}{\sum_{N_i} A_{N_i}}$$

mit

Bezeichnung	Bedeutung
C_{P_i}	empfangene Energiedichte des i-ten Punktes
N_i	Nachbarflächen; Flächen, die den i-ten Punkt umgeben
B_{N_i}	Energiedichte der Nachbarfläche N_i
A_{N_i}	Flächeninhalt der Nachbarfläche N_i

Da mit den beschriebenen Formeln die *empfangene* Energiedichte eines Punktes berechnet wird, ist es gleichgültig, welche der beteiligten Flächen man dem Punkt zuordnet. Diese Tatsache hat einige Vorteile, wie an folgendem Beispiel zu erkennen ist.

Man stelle sich ein Schachbrett vor, das durch diverse benachbarte schwarze und weiße Flächen modelliert ist. Berechnet man die *ausgestrahlte* Energiedichte der Spielfeld-Eckpunkte, so ergibt sich für ein und denselben Punkt je nach Spielfeldfarbe ein verschiedener Wert. Die *empfangene* Energiedichte ist aber für diesen Punkt für alle ihn umgebenen Spielflächen identisch. Natürlich muß bei der Darstellung die empfangene Energiedichte noch mit der Reflektivität (=Flächenfarbe) der jeweils darzustellenden Fläche multipliziert werden. Die Speicherung des Ergebnisses erfolgt in den Punkten selbst.

Algorithmus 18.7 (Wandele Energien in Punktfarben um)

FÜR alle Flächen, die Element sind:
FÜR alle Eckpunkte der Fläche:
Setze Flächensumme auf den Flächeninhalt dieser Fläche.
*Setze Energiesumme auf Energiedichte*Flächeninhalt dieser Fläche*
Bestimme anhand der Kanten, an denen der Punkt beteiligt ist, ..
.. die Nachbarflächen, die diesen Punkt umgeben.
FÜR alle diese Nachbarflächen:
Addiere den Flächeninhalt der Nachbarfläche zur Flächensumme.
Addiere Energiedichte · Flächeninhalt der Nachbarfläche ..
.. zur Energiesumme.
ENDE FÜR alle diese Nachbarflächen.
Empfangene Energiedichte des Punktes=Energiesumme/Flächensumme.
ENDE FÜR alle Eckpunkte der Fläche.
ENDE FÜR alle Flächen.

Die Schleife über alle Eckpunkte aller Flächen, die Element sind, ist in `BerechnePunktFarbenAusEnergien` realisiert, die dann für diese Eckpunkte die Prozedur `MittlereBeteiligteFlaechen` aufruft. In ihr findet dann der Rest der Berechnung statt.

`MittlereBeteiligteFlaechen` wird noch an zwei anderen Stellen in `PRIM` verwendet. Bei scharfen Punkten kann es je nach Nachbarschaftsverhältnissen vorkommen, daß dessen durch die Radiosity-Beleuchtung erzeugte Farbe aus verschiedenen (Nachbar-)Flächen generiert wird, je nachdem zu welcher Fläche der Punkt im speziellen Fall als Eckpunkt gedeutet wird.

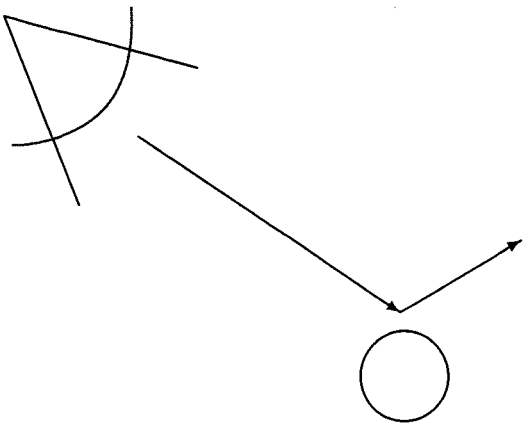
Die Darstellung der Szenen benötigt die empfangene Energiedichte sowie die (diffuse) Reflektivität der Fläche. Das Produkt aus beiden ist dann die diffuse Farbe des Eckpunktes. Nun kann aber zusätzlich noch eine „normale" Beleuchtung erfolgen, um z.B. eine Annäherung für die spiegelnde Reflexion des Lichtes zu erreichen. Die Ergebnisse beider Beleuchtungsmodelle können dann vermischt werden. Auf die gleiche Art ist es auch möglich, der Radiosity-Iteration einen Raytracing-Durchlauf anzuschließen und dann die Ergebnisse dieser beiden Verfahren zu mischen. Einer Kombination von Radiosity und Raytracing steht also nichts mehr im Wege.

Will man Raytracing oder auch Phong-Shading auf die durch Radiosity vorbeleuchtete Szene anwenden, so ist es allerdings nötig, die diffuse Farbe der Eckpunkte, also das Ergebnis der Radiosity-Berechnung, innerhalb der Polygone zu interpolieren.

IV

Raytracing

Rolf-Dieter Klein



19. Raytracing

Jetzt wird es noch „realer". Bisher wurden noch keine scharfen Schatten, Spiegelungen und transparente Objekte wie zum Beispiel Linsen verwirklicht. Dies ist mit den bisher besprochenen Verfahren recht aufwendig, aber es gibt eine Methode, mit der sich der Programmaufwand in Grenzen hält: das „Raytracing" oder zu deutsch: die „Strahlverfolgung".

Mit diesem Verfahren ist es auch möglich, Grundobjekte analytisch zu behandeln und so zum Beispiel eine Kugel exakt darzustellen und nicht nur durch Annäherung von Facetten. Raytracing hat aber auch einen Nachteil: es ist sehr rechenintensiv, und daher dauert die Berechnung eines Bildes wesentlich länger als mit den zuvor beschriebenen Verfahren, denn für jeden Bildpunkt sind umfangreiche Berechnungen notwendig.

19.1 Die Strahlverfolgung

19.1.1 Einführung

Die Idee beim Raytracing ist es, den Verlauf der Lichtstrahlen in einer Szene zu verfolgen und dadurch die Szene so zu berechnen, wie sie in der Natur erzeugt wird. Man sich das ganz einfach folgendermaßen vorstellen:

Licht kommt aus Lichtquellen. Also schickt man Lichtstrahlen von den Lichtquellen aus und berechnet den Strahlverlauf solange, bis das Auge des Beobachters getroffen wird. Treffen die Lichtstrahlen auf Objekte, so werden sie je nach Materialeigenschaft gestreut, erhalten eine andere Farbe oder werden absorbiert. Schließlich gelangen einige Lichtstrahlen zum Auge. Wenn man sich einen Bildschirm im Raum aufgehängt vorstellt, muß man jetzt nur die Lichtstrahlen betrachten, die auf den Bildschirm treffen und dort Farbe und Helligkeit bestimmen. Abb. 19.1 zeigt eine Szene mit ein paar eingezeichneten Lichtstrahlen.

Der Lichtstrahl 1 führt von der Lampe direkt zum Auge und hinterläßt auf dem eingezeichneten Bildschirm einen hellen Punkt. Der Strahl 2 trifft auf das Sofa, wird dort reflektiert und gelangt als Strahl 3 zum Auge und auf den Bildschirm. Bei Strahl 4 ist das anders. Er gelangt an die Decke und wird in unserem Fall von einer dunklen Stelle absorbiert. Strahl 5 gelangt zum Boden und wird dort stark gestreut. Das bedeutet, daß eine Vielzahl von neuen Lichtstrahlen er-

Abb. 19.1 Verfolgung der Lichtstrahlen von der Lichtquelle zum Auge

zeugt werden. Die zweite Lichtquelle an der Decke sendet entsprechend Strahlen aus, die ebenfalls berücksichtigt werden müssen. Einer davon (Strahl 6) gelangt zur Wand und wird dort wieder gestreut. Sie können sich nun unschwer vorstellen, daß auf diese Weise in kürzester Zeit eine Unmenge von Lichtstrahlen erzeugt werden, deren Verlauf berechnet werden müßte. Dabei erreicht nur ein Bruchteil der Strahlen wirklich das Auge des Beobachters. Der Rechenaufwand steigt dabei aber ins Unermeßliche.

Es gibt aber noch eine andere Möglichkeit, Raytracing durchzuführen. Dazu werden nicht die Strahlen von der Lichtquelle zum Beobachter berechnet, sondern umgekehrt die Strahlen vom Auge zur Lichtquelle verfolgt. Es sind dazu, zunächst vom Auge ausgehend, nur soviele Strahlen nötig, wie Bildpunkte auf dem Bildschirm sichtbar sind. Strahlen, die nie das Auge erreichen, müssen nicht berechnet werden. Drei verschiedene Fälle können dabei auftreten.

1. Der Strahl trifft auf eine Lichtquelle. D.h. man kann die Lichtquelle direkt sehen und ein heller Punkt mit der Farbe der Lichtquelle wird auf den Bildschirm gezeichnet.
2. Der Strahl trifft auf den Hintergrund. Hier verwendet man eine beliebige Hintergrundfarbe für den Bildschirm, meist blau (wie der Himmel).
3. Ein Objekt wird getroffen. In diesem Fall muß die Objektfarbe bestimmt werden. Diese setzt sich aber aus unterschiedlichen Komponenten zusammen, wie Sie schon von früheren Kapiteln her wissen. Bei einem matten Objekt ist die Farbe abhängig von der Beleuchtung, diese hängt aber auch von anderen Objekten ab. Ist ein anderes Objekt zwischen dem Objekt und der Lichtquelle, so wirft dieses einen Schatten. Deshalb ist es notwendig, einen Strahl zu jeder Lichtquelle zu senden, um so festzustellen, ob sich ein Hindernis dazwischen befindet. Wenn nicht, so kann die Lichtquelle zur Berechnung der Objektfarbe verwendet werden.
   Objekte können auch spiegeln. In diesem Fall muß zusätzlich ein Strahl in die Reflexionsrichtung geschickt werden. Er bestimmt, ob z.B. weitere

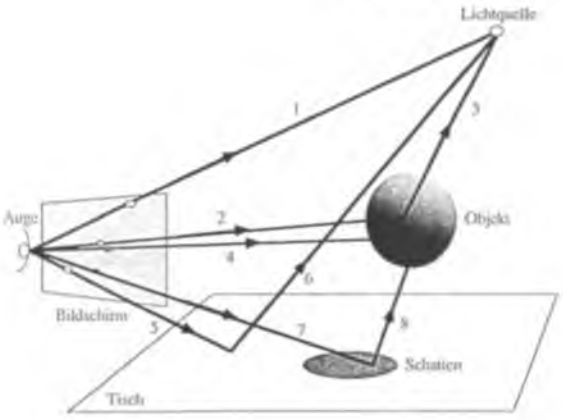


Abb. 19.2 Unterschiedliche Strahlen werden ausgesandt

Objekte sichtbar sind und als Reflexion auf der Oberfläche auftauchen. Hier wird das Raytracing rekursiv.
Das gleiche gilt für transparente Objekte. Man schickt dazu einen Strahl in die Richtung der Brechung und verwendet dann die zurückgelieferte Lichtinformation zur Berechnung der Gesamtbeleuchtung.

Die Berechnung der Beleuchtung aus den Teilkomponenten geschieht wie schon in den vorherigen Kapiteln. Neu ist hier die Möglichkeit, Schattenwurf, Objekte, die sich in Objekten spiegeln, und transparente Objekte darzustellen. Abb. 19.2 zeigt ein Beispiel dazu.

Strahl 1 trifft die Lichtquelle direkt. Strahl 2 trifft die Kugel. Nun muß bestimmt werden, ob die Lichtquelle von diesem Punkt aus sichtbar ist. Dazu wird ein weiterer Strahl, der sogenannte Schattenstrahl, in Richtung Lichtquelle geschickt. In diesem Fall gibt es keine Hindernisse. Daher wird das Objekt von der Lichtquelle beleuchtet und Helligkeit und Farbwerte können berechnet werden. Bei Strahl 4 sieht es anders aus. Er trifft ebenfalls das Objekt, jedoch liegt die Kugel selbst im Weg, wenn man versucht, einen Schattenstrahl zur Lichtquelle zu schicken. Dies gilt für die gesamte von der Lichtquelle abgewendete Seite der Kugel. Sie liegt im eigenen Schatten. Man wird daher nur die Umgebungshelligkeit zur Berechnung der Farbwerte heranziehen.

Strahl 5 trifft auf die Tischoberfläche. Ein Strahl in Richtung Lichtquelle trifft auf keine Hindernisse. Den Tisch könnte man dann entsprechend der Tischfarbe einfärben. Strahl 7 trifft ebenfalls den Tisch, jedoch der Schattenstrahl 8 in Richtung Lichtquelle findet die Kugel als Hindernis. Somit wird der Tisch nur von der Umgebungshelligkeit beleuchtet. Er erscheint daher an dieser Stelle dunkler. Hier wird also ein Schatten sichtbar sein.

Mit dem beschriebenen Verfahren erhält man schon recht ansprechende Bilder. Die Schatten sind jedoch nur ganz hart, da zum Beispiel der Durchmesser einer Lichtquelle bei diesem einfachen Verfahren nicht berücksichtigt wird. In einem späteren Abschnitt werden Sie aber Verbesserungen kennenlernen, mit

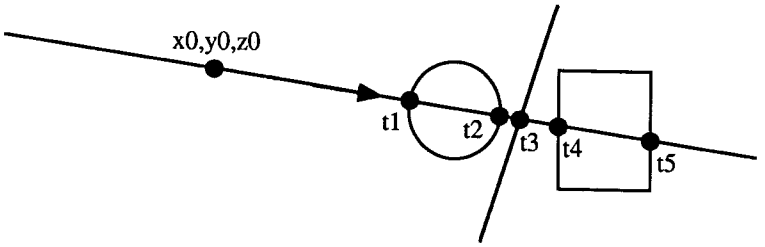


Abb. 19.3 Ein Sichtstrahl schneidet Objekte. Die Auswertung der Schnittgleichungen.

denen auch Halbschatten und andere Effekte möglich sind. Natürlich sind diese Verfahren auch wieder rechenzeitaufwendig.

Dem Buch liegt ein Beispiel-Raytracer (`RDKRAY`) bei, mit dem Sie auch eigene Erfahrungen sammeln können. Er ist in Turbo-Pascal geschrieben und Sie können neue Funktionen leicht hinzufügen. Bitte beachten Sie die Datei `READ.ME` im entsprechenden Unterverzeichnis, die aktuelle Hinweise zur Bedienung gibt. Doch nun ans Werk, und wir fangen zunächst einmal mit den mathematischen Grundlagen an.

19.1.2 Lichtstrahlen

Beim Raytracer besteht die Hauptaufgabe darin, Lichtstrahlen mit Objekten zu schneiden. Daher werden wir zunächst die mathematische Darstellung eines Strahls betrachten. Mathematisch gesehen handelt es sich um Geraden. Diese werden am einfachsten mit der Geradengleichung wie folgt beschrieben:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix} \cdot t$$

Dabei zeigt der Vektor $\begin{pmatrix} x_0 & y_0 & z_0 \end{pmatrix}^T$ zum Ursprung des Beobachters und $\begin{pmatrix} x_d & y_d & z_d \end{pmatrix}^T$ ist der Vektor für die Blickrichtung. Dieser Vektor wird normiert, so daß gilt:

$$\sqrt{x_d \cdot x_d + y_d \cdot y_d + z_d \cdot z_d} = 1.$$

Für $t > 0$ erhält man einen Punkt auf der Geraden, der vor dem Beobachter liegt, und für $t < 0$ einen hinter dem Beobachter. Diese Gleichung kann man nun in andere Gleichungen zur Beschreibung der einzelnen Objekte einsetzen und erhält so Werte für t als Lösung. Der kleinste Wert für t, der größer als Null ist, bestimmt den Schnittpunkt für dasjenige Objekt, welches als erstes geschnitten wird und das damit für den Beobachter sichtbar ist.

Abb. 19.3 zeigt ein Beispiel mit einigen Objekten. Wenn die Schnittgleichungen ausgewertet werden, erhält man die Lösungen $t_1, t_2, \ldots, t_5$. Der

Schnittpunkt t_1 liegt am nächsten beim Beobachter, daher ist die dazugehörige Oberfläche sichtbar. Den Schnittpunkt erhält man, wenn t_1 in die Geradengleichung eingesetzt wird. Wenn der Schnittpunkt berechnet ist, muß die Objektfarbe bestimmt werden. Dazu kann es nötig sein, weitere Lichtstrahlen auszusenden, wenn das Objekt zum Beispiel eine spiegelnde Oberfläche besitzt.

Doch zunächst einmal muß, ausgehend vom Beobachter, ein Strahl durch die Bildschirmfläche geschickt werden, um so die Szene abzutasten. Damit der Raytracer eine Szene berechnen kann, in der der Beobachter frei beweglich ist, sind zusätzliche Parameter für den Raytracer notwendig:

1. Der Beobachterstandort (mit `viewfrom` bezeichnet).
2. Ein Punkt in der Szene, wohin geblickt wird (mit `viewat` bezeichnet).
3. Ein Vektor, der angibt, wo oben ist (mit `viewup` bezeichnet).
4. Der Blickwinkel `viewangle`, der für die Berechnung der Perspektive relevant ist.

Zusätzlich wird die Bildschirmauflösung in x- und y-Richtung benötigt. Daraus läßt sich die Abtastdichte bestimmen.

Zunächst ein paar Definitionen von Datenstrukturen, wie sie in unserem Raytracer häufig verwendet werden:

```
vektor =  RECORD          (* Vektor in allgemeiner Form *)
  x,y,z : DOUBLE;
END;

colorrec = RECORD         (* Farbe im Bereich 0..1 ist zulaessig *)
  r,g,b : DOUBLE;
END;

rayrec = RECORD           (* Ein Strahl wird durch  *)
  point : vektor;         (* einen Punkt *)
  direction : vektor;     (* und eine Richtung definiert *)
END;

viewrec = RECORD          (* Beobachter-Eigenschaften *)
  viewfrom : vektor;
  viewat : vektor;
  viewup : vektor;
  viewangle : DOUBLE;
  viewdist : DOUBLE;
END;
```

Der Raytracing-Algorithmus läßt sich dann wie folgt in ein Programm fassen (verkürzte Version aus dem Beispiel `RDKRAY.PAS` der beigefügten Diskette):

```
PROCEDURE doraytrace;
VAR ... deklarationen...
BEGIN
  (* Berechnen zusaetzlicher Vektoren *)
  vekcopy(view.viewup,upvec);                (* wo ist oben ... *)
  veksub(view.viewat,view.viewfrom,viewvec);   (* Richtung Ziel *)
  veknormalize(viewvec);                  (* muss normiert werden *)
  vekcross(view.viewup,viewvec,leftvec);  (* Daraus Vektor links *)
  veknormalize(leftvec);                   (* abl. und normieren *)
  veknormalize(upvec);                    (* ebenfalls normieren *)

  (* Breite des Bildes berechnen *)
  width := sin(view.viewangle)/cos(view.viewangle);       (* tan *)
  vekscale(-width,upvec);               (* Die Vektoren skalieren *)
  vekscale(-width,leftvec);            (* und Vorzeichen anpassen *)
  vekcopy(view.viewfrom,ray.point);   (* Start der Sichtstrahlen *)

  (* hier beginnt nun die Raytracer-Riesenschleife *)
  FOR y:=0 TO yres-1 DO BEGIN     (* alle Zeilen des Bildschirms *)
    FOR x:=0 TO xres-1 DO BEGIN   (* und alle Spalten bearbeiten *)
      xlen := 2.0 * x / xres - 1.0;       (* Im Bildschirmraster *)
      ylen := 2.0 * y / yres - 1.0;   (* abtasten z. Sichtstrahl *)
      vekcomb(xlen,leftvec,ylen,upvec,ray.direction);
      vekadd(ray.direction,viewvec,ray.direction);
      veknormalize(ray.direction); (* im record ray alles bereit *)
      trace(0,1.0,ray,color);       (* RAYTRACER HIER aufgerufen *)
      (* Punkt mit berechneter Farbe hier setzen *)
      setdot(x,y,color);
    END;
  END;
END;
```

Die eigentliche Raytrace-Routine ist in `trace` verborgen. Sie berechnet die Farbe für den ausgesandten Sichtstrahl. Als zusätzliche Parameter werden der Level, hier 0, und ein Gewichtungsfaktor, hier 1.0, angegeben. Global sind weitere Werte vorhanden, so die maximale Iterationstiefe, für welche die Strahlen noch verfolgt werden sollen. Damit wird bestimmt, wie oft ein Strahl reflektiert bzw. gebrochen werden kann. Je größer die maximale Iterationstiefe ist, desto rechenaufwendiger wird das Verfahren. Eine Tiefe von 4 bis 5 ist meist jedoch nicht mehr von höheren Tiefen unterscheidbar. Denn wer achtet schon auf eine Kugel, die sich in einer Kugel spiegelt, die sich in einer Kugel spiegelt, die sich in einer Kugel spiegelt, ..., zumal die Farbauflösung des Bildschirms eine Unterscheidung unmöglich macht.

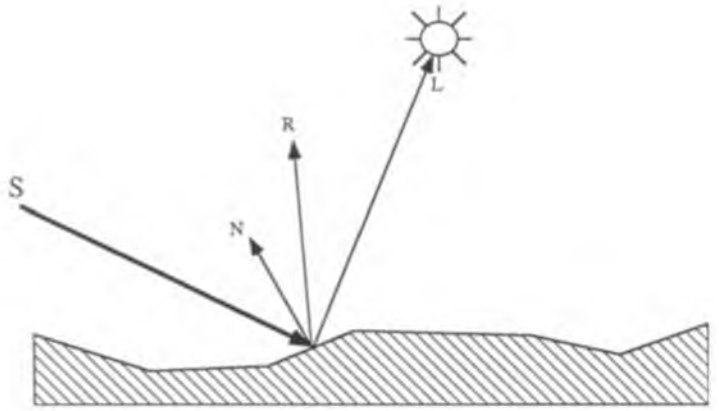


Abb. 19.4 Sichtstrahl, Normale, Reflexionsstrahl und Lichtstrahl (auch Schattenstrahl genannt)

19.1.3 Schatten und Spiegelungen

Hat man einen Schnittpunkt berechnet, so muß die dazugehörige Oberfläche mit ihren Eigenschaften bestimmt werden. Um welche Oberfläche es sich handelt, kann man schon beim Berechnen der Schnittpunkte in einer Datenstruktur festhalten (siehe auch das Beispielprogramm `RDKRAY.PAS` auf der mitgelieferten Diskette). Für das Raytracing von besonderer Bedeutung ist der Normalenvektor auf dieser Oberfläche am Schnittpunkt. Damit lassen sich zum Beispiel Spiegelungen berechnen. Ferner muß man einen Vektor zu jeder vorhandenen Lichtquelle bestimmen, um festzustellen, ob sich der Punkt im Schatten befindet. Andernfalls muß die Lichtquelle bei der Berechnung des Farbwertes berücksichtigt werden. Reflektierte Strahlen werden außerdem weiterverfolgt und liefern den Farbwert von anderen Objekten oder dem Hintergrund zurück (sie werden auch beim Beleuchtungsmodell benötigt). Abb. 19.4 zeigt ein Schema.

Der eingehende Sichtstrahl trifft auf beliebig geformte Oberflächen. Für den Auftreffpunkt gibt es einen Normalenvektor, der sich anhand der Oberflächeneigenschaften ermitteln läßt, und hier als gegeben betrachtet wird. Zu berechnen sind nun der Reflexionsvektor, wenn die Oberfläche spiegelnde Eigenschaften hat, und der Lichtvektor für jede vorhandene Lichtquelle. Der Richtungsvektor zur Lichtquelle ist ganz einfach zu berechnen:

$$L = V_{\text{treff}} - V_{\text{licht}}.$$

Der Startpunkt ist V_{treff}. Damit ist die Geradengleichung durch (V_{treff}, L) gegeben. Den Vektor L muß man allerdings zuvor noch normieren.

Die Berechnung des Reflexionsvektors ist etwas aufwendiger. Es gilt: Einfallswinkel gleich Ausfallwinkel. Abb. 19.5 zeigt die geometrische Beziehung. Der Reflexionsvektor kann dann mit Hilfe der geometrischen Zeichnung nach Abb. 19.6 berechnet werden. Dabei ist wichtig zu wissen, daß der Richtungsvektor S des Sichtstrahls sowie die Senkrechte N die Länge 1 haben, da sie

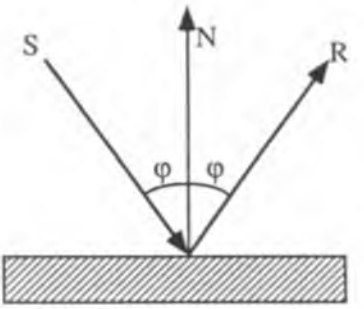


Abb. 19.5 Das Reflexionsgesetz

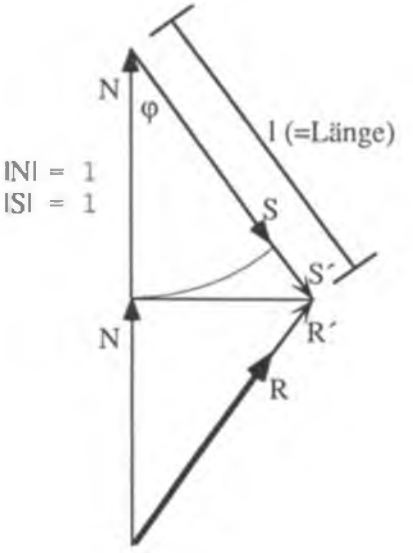


Abb. 19.6 Die Ermittlung des Reflexionsvektors aus Sichtstrahl und Normale

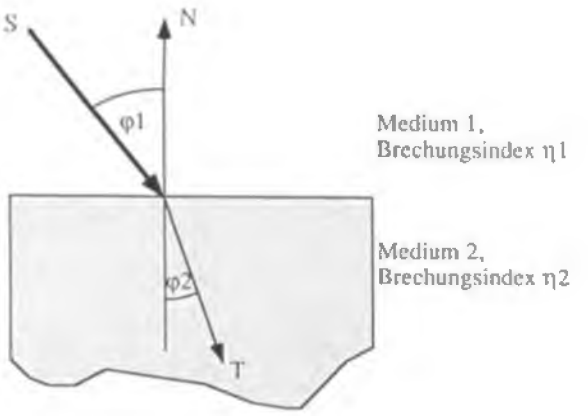


Abb. 19.7 Das Brechungsgesetz

normiert sind. Es gilt dann (vergleichen Sie auch mit Kap. 10):

$$\begin{aligned} l &= \frac{1}{\cos\phi} \\ \cos\phi &= (S \cdot N) \\ R' &= 2 \cdot N - S' \\ R' &= 2 \cdot N - l \cdot S \\ R' &= 2 \cdot N - \frac{1}{(S \cdot N)} \cdot S \\ R &= \text{norm}(R') \end{aligned}$$

Damit läßt sich R relativ einfach bestimmen und die Geradengleichung für den Reflexionsstrahl ist dann durch (V_{treff}, R) bestimmt.

19.1.4 Brechungen

Bei Objekten, die transparent sind, muß auch ein das Objekt durchquerender Lichtstrahl berechnet werden. Dabei wird der Lichtstrahl je nach Materialeigenschaft unterschiedlich gebrochen. Abb. 19.7 zeigt den Vorgang. Der Sichtstrahl trifft auf die Oberfläche im Winkel ϕ_1. Er wird dann gebrochen und läuft im Material mit dem Winkel ϕ_2 weiter. Die Abhängigkeit von ϕ_1 zu ϕ_2 wird durch das Brechungsgesetz (formuliert von Willebrord von Roijen Snell, im Jahr 1621) beschrieben:

$$\frac{\sin\phi_1}{\sin\phi_2} = \frac{\eta_2}{\eta_1}$$

Dabei wird durch η_1 der Brechungsindex von Medium 1 gegenüber dem Vakuum und durch η_2 der Brechungsindex von Medium 2 gegenüber dem Vakuum beschrieben. Wenn das Medium 2 einen höheren Brechungsindex hat als Medium 1, dann wird der Strahl zur Normalen hin gebrochen. Der Brechungsindex ist im allgemeinen abhängig von der Wellenlänge, also der Farbe des Lichtes.

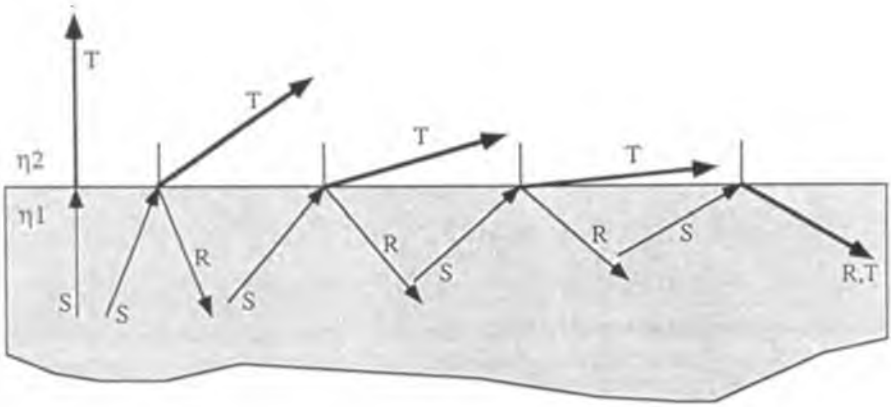


Abb. 19.8 Die Totalreflexion

Damit Sie unterschiedliche Materialien beim Raytracing berücksichtigen können, hier eine kleine Tabelle verschiedener Brechungsindizes:

Medium	Brechungsindex
Luft (760 Torr, 20° C)	1.003
Wasser	1.333
Äthylalkohol	1.36
Quarz	1.46 ... 1.50
Benzol	1.501
Kronglas	1.51
Steinsalz	1.544
Flintglas	1.61
Schwefelkohlenstoff	1.628
Diamant	2.417

Wenn das Licht von einem dichteren Medium zu einem weniger dichten gelangt, so kann eine Totalreflexion eintreten. Abb. 19.8 zeigt ein Schema. Der von unten aus einem dichteren Medium austretende Sichtstrahl wird an der Oberfläche gebrochen, aber auch reflektiert. Wenn der Strahl zu flach eintritt, so gibt es nur noch den reflektierten Strahl, also Totalreflexion. Dies muß im Raytracing-Programm durch eine Abfrage berücksichtigt werden.

Zur Berechnung des Brechungsstrahls kann man ähnlich vorgehen wie schon beim Reflexionsstrahl. Der Strahl wird aus einer Kombination der beiden Vektoren S und N zusammengesetzt. Damit lautet die Formel allgemein:

$$T = \alpha \cdot S + \beta \cdot N$$

α und β müssen jetzt noch bestimmt werden. Es ergibt sich folgende Lösung:

$$\begin{aligned} \alpha = \eta &= \frac{\eta_1}{\eta_2} \\ c_1 &= -(I \cdot N) \\ c_2 &= 1 - \eta^2 \cdot (1 - c_1^2) \\ T &= \eta \cdot I + (\eta \cdot c_1 - \sqrt{c_2}) \cdot N \end{aligned}$$

Wenn c_2 kleiner als Null ist, dann liegt Totalreflexion vor, und die Berechnung für T entfällt ($\sqrt{c_2}$ wäre imaginär). Die Berechnung der Brechung ist beim Raytracing schon ein Zeitfaktor, denn sie muß für jeden Punkt der brechenden Oberfläche durchgeführt werden, der vom Beobachter zu sehen ist.

19.1.5 Der Raytrace-Algorithmus

Hier soll das bisher beschriebene nochmals zusammengefaßt dargestellt werden. Dabei ist wichtig zu wissen, daß man für diesen Algorithmus von den Objekten selbst nur den Normalenvektor benötigt.

Mit den bisher behandelten Algorithmen läßt sich der Raytracer fast fertig formulieren. Der Schnitt-Algorithmus ist dabei noch in der Prozedur `intersect` verborgen, mit der wir uns später noch beschäftigen werden. Für die Schattenbestimmung wird im Raytracer eine Funktion `notinshadow` verwendet. Sie arbeitet wie folgt:

```
FUNCTION notinshadow(ray : rayrec;
             VAR hitdata : isectrec; tmax : DOUBLE) : BOOLEAN;
BEGIN
  IF (intersect(worldptr[0],ray,hitdata,tmax)) THEN BEGIN
    notinshadow := FALSE;
  END ELSE BEGIN
    notinshadow := TRUE;
  END;
END;
```

In `worldptr[0]` ist die Objektbeschreibung aller in der Szene vorkommenden Objekte verborgen. Sie wird als Parameter an `intersect` übergeben. `ray` ist in diesem Fall der Schattenstrahl. Die Datenstruktur `hitdata` wird hier nicht weiter ausgewertet, sie enthält Informationen über das getroffene Objekt, die aber für die Schattenberechnung nicht weiter interessant sind. Die Variable `tmax` gibt den bisher maximalen t-Wert für den Schnittpunkt an. Er wird für die Schattenberechnung mit dem Abstand zwischen Lichtquelle und Ausgangspunkt belegt. Schnitte hinter der Lichtquelle sind uninteressant und werden so ignoriert. Nur Schnittpunkte zwischen dem Ausgangspunkt auf der Oberfläche und der Lichtquelle bedeuten Schatten. Die Routine `intersect` liefert den Wert `TRUE`, wenn dies der Fall ist.

Als nächstes ist die Prozedur für den Reflexionsstrahl an der Reihe. Hier werden auch die Unterprogramme für Vektorrechnung verwendet. In `RDKRAY.PAS` finden Sie die vollständigen Quellen.

```
PROCEDURE speculardirection(i, n : vektor; VAR r : vektor);
BEGIN
  vekcomb(1.0 / abs(vekdot(i,n)),i,2.0,n,r);
  veknormalize(r);
END;
```

Die Prozedur berechnet aus dem eingehenden Strahl S (Sichtstrahl, im Programm die Variable i) und der Normalen N den Reflexionsvektor R, wobei die Routine `vekcomb` folgende Formel benutzt:

```
r := 1/abs(vekdot(i,n) * i + 2.0 * n.
```

Die Brechungsberechnung erfolgt analog dazu. Die Funktion `transmissiondirection` liefert den Wert `TRUE`, falls eine Brechung vorliegt, sonst `FALSE`. Hier werden zwei weitere Parameter übergeben, nämlich Pointer auf die Oberflächeneigenschaften (Brechungsindizes) mit `surf1p` und `surf2p`:

```
FUNCTION transmissiondirection(surf1p,surf2p :surfpp;
            raydir : vektor; normal: vektor;
            VAR direction : vektor):BOOLEAN;
VAR
  n1, n2, eta, c1, cs2 : DOUBLE;
BEGIN
  IF surf1p <> NIL THEN n1 := surf1p^.ior ELSE n1 := 1.0;
  IF surf2p <> NIL THEN n2 := surf1p^.ior ELSE n2 := 1.0;
  eta := n1 / n2;
  c1 := -vekdot(raydir,normal);
  cs2 := 1.0 - eta * eta * (1.0 - c1*c1);
  IF cs2 < 0.0 THEN BEGIN
    transmissiondirection := FALSE; (* totalreflexion *)
  END ELSE BEGIN
    vekcomb(eta,raydir,eta*c1-sqrt(cs2),normal,direction);
    transmissiondirection := TRUE;
  END;
END;
```

Wenn kein Pointer verwendet wird, so wird als Brechungsindex 1.0 eingesetzt. Wenn die Funktion `FALSE` liefert, liegt eine Totalreflexion vor und der Brechungsstrahl ist undefiniert.

Im Programmverlauf wird die Prozedur `trace` rekursiv aufgerufen. Daher muß in Pascal eine Vorwärts-Deklaration erfolgen:

```
PROCEDURE trace(level : INTEGER; weight : DOUBLE;
             ray : rayrec; VAR color : colorrec;
             VAR nr : LONGINT); forward;
```

Der Index `nr` liefert dabei eine Information über die Anzahl der Strahlen ab, die im Verlauf der Ausführung von `trace` berechnet wurden. Diese Information wird vom Programm `RDKRAY` am Schluß ausgegeben. `Level` ist die Iterationstiefe des Raytracers und `weight` ein Gewichtungsfaktor, der intern als Abbruchkriterium verwendet wird. Dieser wird am Anfang mit 1.0 belegt.

Das Programmteil `shade` verwendet jetzt alle bisher definierten Routinen und führt die Berechnung der Objektfarbe aus. Es wird hier zur Übersicht in vereinfachter Form dargestellt. Die Vektoren `hpoint` und `normal` werden dabei vom Auftreffpunkt und dessen Normalen bestimmt. `raydir` ist der Richtungsvektor des einfallenden Strahls und `hitdata` enthält alle Informationen des

getroffenen Objekts. Daraus wird die Farbe `col` abgeleitet, als ein Record mit den Komponenten `r,g` und `b`.

```
PROCEDURE shade(level : INTEGER; weight : DOUBLE;
                hpoint : vektor; normal : vektor;  raydir : vektor;
                hitdata : isectrec; VAR col: colorrec);
VAR
  i          : INTEGER;
  surfp      : surfpp;
  lght,sperg : vektor;
  tray       : rayrec;
  t,spec     : DOUBLE;
  diff,tcol  : colorrec;
  flag       : BOOLEAN;
BEGIN
```

Die Farbe wird zunächst auf „schwarz" initialisiert und im Verlauf der Berechnung immer genauer bestimmt.

```
  col.r := 0.0;
  col.g := 0.0;
  col.b := 0.0;
  surfp := (hitdata.primptr)^.surfptr;
  (* Oberflaechen-Eigenschaften in surfp festhalten *)
  (* Nun erfolgt der Schattentest fuer alle Lichtquellen *)
```

Es werden alle vorhandenen Lichtquellen untersucht. Zunächst wird der Lichtvektor berechnet.

```
  FOR i:= 1 TO numberoflights DO BEGIN
    veksub(lights[i].lightpos,hpoint,lght);     (* Lichtvektor *)
    IF (vekdot(normal,lght) >= 0.0) THEN BEGIN (* Licht davor *)
```

Nur wenn das Skalarprodukt zwischen Lichtvektor und Normalen größer als Null ist, ist die Lichtquelle von der Oberfläche aus prinzipiell sichtbar, sonst wird sie durch die Oberfläche selbst verdeckt. Hier wird eine diffuse Transparenz nicht berücksichtigt, was aber prinzipiell auch möglich wäre.

```
      t := fveknormalize(lght);
      tray.point := hpoint;
      tray.direction := lght;
      nshadows := nshadows + 1;  (* Fuer Statistik mitzaehlen. *)
```

Es erfolgt als nächstes die Prüfung, ob sich weitere Objekte zwischen Oberfläche und Lichtquelle befinden:

```
      IF (notinshadow(tray,hitdata,t)) THEN BEGIN
```

Die Lichtquelle erreicht das Objekt und trägt damit zur Oberflächenfarbe bei. Hier eine einfache Möglichkeit, dies zu berücksichtigen. Andere komplizierte Beleuchtungsmodelle sind natürlich möglich. Die Oberfläche hat in unserem Beispiel in `kdiff` einen Faktor gespeichert, mit der die Lichtquellenfarbe die Oberflächenfarbe beeinflußt.

```
diff.r := vekdot(normal,lght) * surfp^.kdiff
          * lights[i].color.r;
diff.g := vekdot(normal,lght) * surfp^.kdiff
          * lights[i].color.g;
diff.b := vekdot(normal,lght) * surfp^.kdiff
          * lights[i].color.b;
```

Die Farben werden addiert und die Oberflächenfarbe multipliziert.

```
col.r := col.r + surfp^.color.r * diff.r;
col.g := col.g + surfp^.color.g * diff.g;
col.b := col.b + surfp^.color.b * diff.b;
```

Bei einer glänzenden Oberfläche wird nun eine zusätzliche Farbkomponente berechnet. Diese hängt nicht mehr von der Objektfarbe ab, sondern nur von derjenigen der Lichtquelle.

```
IF (surfp^.shine > rayeps) THEN BEGIN
speculardirection(raydir,normal,sperg);
veknormalize(sperg);
spec := vekdot(sperg,lght);
IF (spec > rayeps) THEN BEGIN
  spec := power(spec,surfp^.shine);
```

Diese wird auch auf die Oberflächenfarbe des Ergebnisses addiert.

```
        col.r := col.r + spec * lights[i].color.r;
        col.g := col.g + spec * lights[i].color.g;
        col.b := col.b + spec * lights[i].color.b;
      END;
    END; (* spec *)
  END; (* Nicht im Schatten *)
END; (* vekdot *)
END; (* for lights *)
```

Als nächstes wird ein Reflexionsstrahl berechnet, jedoch nur, wenn die Oberfläche reflektierend ist und eine Multiplikation mit dem Gewichtungsfaktor noch einen ausreichend großen Wert liefert.

```
tray.point := hpoint;
IF (surfp^.kspec * weight > minweight) THEN BEGIN
```

Dies ist hier der Fall und es wird dann die Richtung des Reflexionsstrahls bestimmt.

```
  speculardirection(raydir,normal,tray.direction);
  veknormalize(tray.direction);
```

Der Raytracer wird hier rekursiv aufgerufen und so der Reflexionsstrahl weiterverfolgt. Der Gewichtungsfaktor wird mit dem Faktor für den spiegelnden Anteil `kspec` multipliziert und übergeben. Außerdem wird die Schachtelungstiefe erhöht.

```
    (*  HIER BEGINNT DIE REKURSION *)
    trace(level+1,surfp^.kspec * weight,tray,tcol,nreflected);
```

Die so ermittelte Farbe wird dann auf das Ergebnis addiert, doch zuvor mit dem Reflexionsfaktor gewichtet.

```
    col.r := col.r + surfp^.kspec * tcol.r;
    col.g := col.g + surfp^.kspec * tcol.g;
    col.b := col.b + surfp^.kspec * tcol.b;
  END;
```

Nun erfolgt die Untersuchung für Materialien mit Brechung.

```
  IF (surfp^.ktran * weight > minweight) THEN BEGIN
```

Bei Brechung wird der Richtungsstrahl bestimmt. Dies geschieht in Abhängigkeit von einem Flag auf unterschiedliche Weise. Unser Raytracer berücksichtigt der Einfachheit halber nur den Übergang von Luft auf ein Medium und umgekehrt, nicht jedoch zwei Medien, die unterschiedlich von Luft sind. Der Brechungsindex hängt dann davon ab, ob der Strahl in das Objekt eintritt oder aus ihm heraus kommt. In `hitdata.enterflag` ist dies festgehalten und wurde von der Schnitt-Routine bestimmt.

```
    IF (hitdata.enterflag) THEN BEGIN
      flag := transmissiondirection(
         NIL,surfp,raydir,normal,tray.direction);
    END ELSE BEGIN
      flag := transmissiondirection(
         surfp,NIL,raydir,normal,tray.direction);
    END;
```

An dieser Stelle wird der Raytracer erneut rekursiv aufgerufen und der Farbwert des ausgesendeten Strahls bestimmt, falls keine Totalreflexion vorliegt (`flag = FALSE`).

```
    IF flag THEN BEGIN (* nur wenn es ok war *)
      trace(level+1,surfp^.ktran * weight,tray,tcol,nrefracted);
```

Der Farbwert wird ebenfalls gewichtet, diesmal mit dem Transparenzfaktor, und zur Ergebnis-Farbe addiert.

```
        col.r := col.r + surfp^.ktran * tcol.r;
        col.g := col.g + surfp^.ktran * tcol.g;
        col.b := col.b + surfp^.ktran * tcol.b;
      END;
    END;
  END;
```

Neben der Schnittberechung, die wir für einzelne Objekte erst in den nächsten Abschnitten besprechen, fehlt noch die Prozedur `trace`.

```
PROCEDURE trace(level : INTEGER; weight : DOUBLE;
                ray : rayrec; VAR color : colorrec;
                VAR nr : LONGINT);
VAR
  hpoint : vektor;
  normal : vektor;
  hitdata : isectrec;  (* Objekt-Datenstruktur *)
BEGIN
  IF (level >= maxlevel) THEN BEGIN
```

Wenn die maximale Rekursionstiefe erreicht ist, liefert der Strahl die Hintergrundfarbe zurück, hier schwarz.

```
    color.r := 0.0;
    color.g := 0.0;
    color.b := 0.0;
  END ELSE BEGIN
```

Sonst wird die Analyse gestartet.

```
    nr := nr + 1; (* Ray Counter erhoehen fuer Statistik *)
    IF (intersect(worldptr[0],ray,hitdata,3.0e20)) THEN BEGIN
```

Wenn ein Objekt gefunden wurde, mit dem sich der Sichtstrahl schneidet, dann wird dieser Schnittpunkt bestimmt. Dazu steht in der Datenstruktur `hitdata` der Wert t für die Geradengleichung.

```
      raypoint(ray,hitdata.t,hpoint);
```

Die nachfolgende Prozedur holt anhand der Objektinformation den Normalenvektor. Dieser kann dort z.B. auch durch Oberflächeneigenschaften wie Bumpmapping zusätzlich verändert sein.

```
      getnormal(hitdata,hpoint,normal); (* Normale holen *)
```

Die Normale muß von der Oberfläche wegzeigen, von der aus der Sichtstrahl auftrifft, damit weitere Berechnungen korrekt durchgeführt werden.

```
      IF (vekdot(ray.direction,normal) >= 0) THEN BEGIN
        vekneg(normal);
      END;
```

Dann kann zum Beispiel der bereits besprochene einfache Shader aufgerufen werden.

```
      shade(level,weight,hpoint,normal,ray.direction,
            hitdata,color);
    END ELSE BEGIN
```

Wenn jedoch kein Objekt getroffen wurde, so erhält der Strahl die Farbe des Hintergrunds, die in einer globalen Variablen steht.

```
      color := backgroundcol;
    END; (* Intersect *)
  END;
END;
```

Jetzt fehlen nur noch die objektspezifischen Routinen zur Schnittpunkt- und Normalenberechnung.

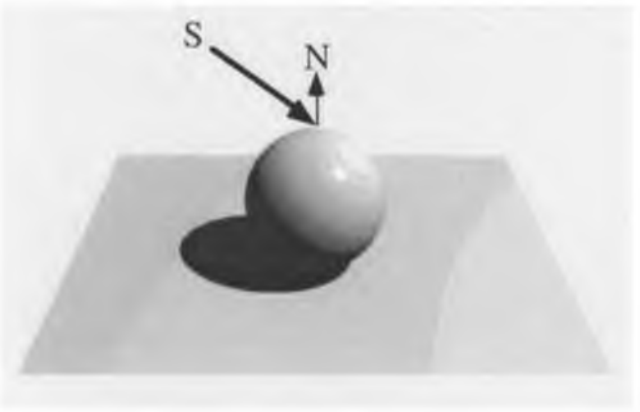


Abb. 19.9 Eine Kugel im Raytracer

19.2 Grundkörper

In diesem Abschnitt werden wir verschiedene Grundkörper behandeln. Im Vordergrund steht dabei die Schnittbestimmung mit dem Sichtstrahl. Für die meisten besprochenen Körper gibt es auch ein Beispiel in dem Programm `RDKRAY.PAS` auf der Diskette. Die entsprechenden Programmabschnitte werden an dieser Stelle ebenfalls besprochen. Ferner gibt es Ergänzungen und Hinweise zum Ausbau des Algorithmus. Die einzelnen Schnittroutinen werden dann im Abschnitt über CSG zusammengefaßt und aufgerufen. Dort ist auch das übergeordnete Programm `intersect` erklärt.

19.2.1 Kugeln

Bei vielen Raytracing-Beispielen findet man Kugeln als Objekte in der Szene (Abb. 19.9). Das kommt daher, daß Kugeln sich besonders einfach mit Geraden schneiden lassen und nur eine einfache Formel ausgewertet werden muß. Neben der Schnittpunktbestimmung wird auch die Normale am Schnittpunkt benötigt. Doch die Berechnung ist nicht sehr kompliziert.

19.2.1.1 Schnitt: Kugel und Strahl

Eine Kugel ist durch ihren Mittelpunkt und einen Radius eindeutig bestimmt (siehe auch Abschnitt 7.2). Die mathematische Gleichung für einen Punkt auf der Oberfläche einer Kugel lautet

$$(X - X_m)^2 + (Y - Y_m)^2 + (Z - Z_m)^2 = R^2,$$

wobei

$\begin{pmatrix} X & Y & Z \end{pmatrix}$	die Koordinaten des Punktes auf der Kugel,
$\begin{pmatrix} X_m & Y_m & Z_m \end{pmatrix}$	die Koordinaten des Kugelmittelpunkts und
R	der Kugelradius ist.

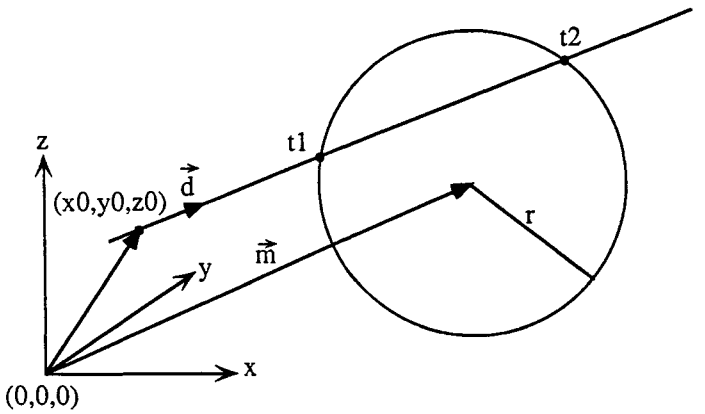


Abb. 19.10 Die Kugel und Schnittpunkte

In diese Formel muß nun die Geradengleichung eingesetzt werden, um daraus die Schnittpunkte t_1 und t_2 zu bestimmen. Abb. 19.10 zeigt den Vorgang. Es entsteht die folgende Gleichung:

$$((x_0 + tx_d) - X_m)^2 + ((y_0 + ty_d) - Y_m)^2 + ((z_0 + tz_d) - Z_m)^2 = R^2.$$

Dies ist eine quadratische Gleichung der Form

$$At^2 + Bt + C = 0$$

und man erhält zwei Lösungen für t. Es ergibt sich für A der Wert 1, da der Richtungsvektor d normiert ist und damit:

$$\begin{aligned} t_1 &= \frac{-B - \sqrt{B^2 - 4C}}{2} \\ t_2 &= \frac{-B + \sqrt{B^2 - 4C}}{2} \\ B &= 2(x_d(x_0 - X_m) + y_d(y_0 - Y_m) + z_d(z_0 - Z_m)) \\ C &= (x_0 - X_m)^2 + (y_0 - Y_m)^2 + (z_0 - Z_m)^2 - R^2 \end{aligned}$$

Wenn der Inhalt der Wurzel negativ wird, so existiert kein Schnittpunkt und die Gerade verläuft außerhalb der Kugel. Für einen Schnitt gibt es zwei Lösungen t_1 und t_2 und im Fall einer Tangente eine Lösung für t.

Bei der Auswertung von t müssen mehrere Dinge beachtet werden:

1. Wenn entweder t_1 oder t_2 kleiner als Null ist, so liegt der jeweilige Schnittpunkt hinter dem Beobachter. Ist nur einer der beiden Werte kleiner Null, so befindet man sich im Inneren der Kugel.
2. Sind beide Werte kleiner Null, so ist die Kugel nicht sichtbar.
3. Wenn der Betrag von t_1 sehr klein, der Wert aber größer Null ist, so liegt der Schnittpunkt unmittelbar vor dem Beobachter.

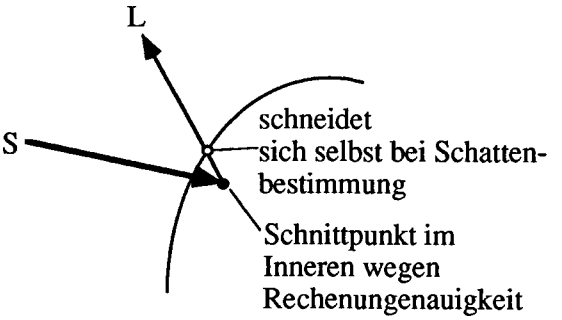


Abb. 19.11 Die Kugel und Rundungsprobleme bei der Schnittpunktberechnung

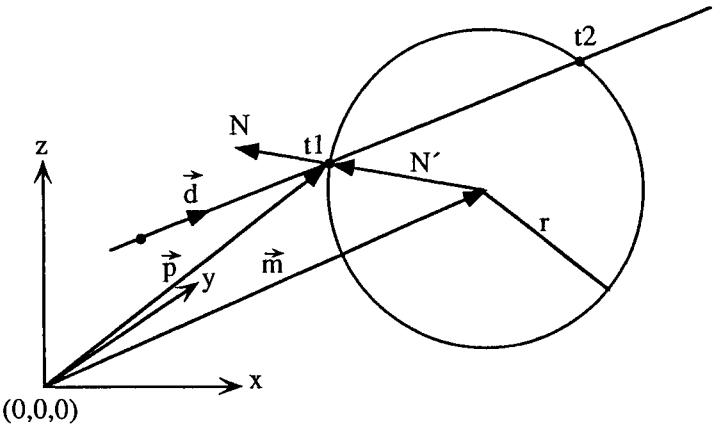


Abb. 19.12 Eine Kugel wird durch den Mittelpunktsvektor und einen Radius definiert.

Den Schnittpunkt von Fall 3 muß man normalerweise auch ignorieren, sonst tritt zum Beispiel bei der Schattenberechnung ein Fehler auf. Abb. 19.11 zeigt, was geschieht. Der Schnittpunkt liegt etwas im Inneren der Kugel, was aufgrund von Rundungsfehlern recht leicht geschehen kann. Ein Schattenstrahl zur Lichtquelle würde nun die Kugel in unmittelbarer Nähe des Schnittpunktes treffen. Damit wirft die Kugel einen Schatten auf sich selbst, was natürlich falsch ist. Wenn man stattdessen t-Werte ignoriert, die sehr klein sind, so kann dieser Fehler nicht auftreten. Man prüft daher $t < \epsilon$, wobei sich $\epsilon = 10^{-6}$ als praktikabler Wert erwiesen hat.

Wenn die Kugel getroffen wurde, so läßt sich der Schnittpunkt leicht berechnen, indem der t-Wert in die Geradengleichung eingesetzt wird. Dann ist es auch möglich, die Normale zu bestimmen. Abb. 19.12 zeigt das dazugehörige Schema. N' läßt sich aus dem Schnittpunkt p und dem Mittelpunkt m berechnen:

$$N' = p - m$$

Durch Normieren von N' erhält man anschließend die Normale N.

Nun zur Programmlösung. Die Funktion `inter_sphere` liefert den Wert `TRUE`, wenn die Kugel vom Sichtstrahl getroffen wurde. Als Parameter erhält die Routine einen Pointer auf die Datenstruktur für die Kugel. Dort ist z.B. der Mittelpunkt und der Radius der Kugel gespeichert, sowie andere Informationen, die später der Shader zur Darstellung benötigt. Die Variable `ray` ist ein Vektor für den Sichtstrahl und nach dem Aufruf steht in der Datenstruktur `hitdata` die Information über die Kugel, wenn diese getroffen wurde.

```
FUNCTION inter_sphere(obj : primpp; ray : rayrec;
                      VAR hitdata : isectrec) : BOOLEAN;
VAR
  b,disc,t,t1,t2 : DOUBLE;
  point : vektor;
  objs : spherepp;
BEGIN (* der Pointer "objs" zeigt auf die Kugelinformationen *)
  objs := obj^.sphereptr;
```

Es erfolgt nun die Schnittpunktermittlung. Die Gleichung wurde noch weiter vereinfacht, z.B. läßt sich der Faktor 2 ausklammern. Die Werte für C und B lassen sich vektoriell formulieren.

```
veksub(objs^.center,ray.point,point);
b := vekdot(point,ray.direction);
disc := b * b - vekdot(point,point) + objs^.radius2;
```

In `radius2` befindet sich das Quadrat des Kugelradius schon in vorberechneter Form. Als nächstes wird geprüft, ob der Inhalt der Wurzel kleiner 0 ist.

```
IF (disc < 0.0) THEN BEGIN
```

Wenn ja, so wird die Kugel nicht geschnitten und weitere Berechnungen können eingespart werden.

```
  inter_sphere := FALSE;
END ELSE BEGIN (* sonst Schnitt moeglich *)
```

Erst hier sind weitere Untersuchungen nötig. Zunächst wird die Wurzel berechnet. Daraus läßt sich der Wert für t_1 und t_2 berechnen. Hier werden beide Werte berechnet, da auch, wenn die Kugel nur t_1 trifft, der Wert t_2 später noch für die CSG (Constructive Solid Geometry, für Mengenbildung) benötigt wird.

```
disc := sqrt(disc);
t := b - disc;
t2 := b + disc;
```

Nun wird untersucht, welcher der t-Werte dem Betrachter am nächsten liegt. Wenn `t < rayeps`, dann ist der Punkt hinter dem Beobachter. `rayeps` ist mit `1E-6` vorbelegt. In diesem Fall werden die beiden gefundenen Schnittwerte getauscht, t erhält den Wert von t_2 und es wird nochmals `t < rayeps` geprüft.

```
    IF t < rayeps THEN BEGIN t1:=t; t:=t2; t2:=t1; END;
    IF t < rayeps THEN BEGIN (* nicht getroffen *)
```

Die Kugel liegt hinter dem Beobachter und kann ignoriert werden.

```
      inter_sphere := FALSE;
    END ELSE BEGIN (* Treffer *)
```

Ein Treffer liegt vor und die Datenstruktur `hitdata` wird mit Werten belegt. Dazu werden beide Schnittwerte t festgehalten.

```
      hitdata.t := t; (* immer der naechste treffer *)
      hitdata.t2 := t2; (* der weiter entferntere *)
```

Für transparente Kugeln und für CSG wird noch die Information benötigt, ob der Strahl in die Kugel eintritt oder aus ihr herauskommt. Dies wird in der Variablen `enterflag` gespeichert.

```
      IF vekdot(point,point)>(objs^.radius2+rayeps) THEN BEGIN
        hitdata.enterflag := TRUE;  (* von aussen nach innen *)
        hitdata.enter2flag := TRUE;
      END ELSE BEGIN
        hitdata.enterflag := FALSE; (* innerhalb der Kugel *)
        hitdata.enter2flag := FALSE;
      END;
```

Nun sind noch Informationen zum getroffenen Objekt festzuhalten. Dazu gibt es zwei Pointer für t_1 und für t_2. Diese Werte sind hier gleich zu besetzen. Erst im Abschnitt zu CSG ist es möglich, auch unterschiedliche Pointer zu erhalten.

```
      hitdata.primptr := obj; (* hier immer das gleiche Objekt *)
      hitdata.prim2ptr := obj;
      inter_sphere := TRUE; (* ok getroffen *)
    END;
  END;
END;
```

Die Berechnung der Normale erfolgt erst dann, wenn sie benötigt wird. Dies ist nur dann der Fall, wenn die Kugel das zum Betrachter nächstliegende Objekt ist.

Als Parameter wird ein Pointer auf die Objekt-Datenstruktur sowie die Schnittinformation und der Schnittpunkt `hpoint` übergeben. Nach dem Aufruf steht in `normal` der normalisierte Normalenvektor.

```
PROCEDURE normal_sphere(obj : primpp; VAR hitdata : isectrec;
                        hpoint : vektor; VAR normal : vektor);
VAR
  objs : spherepp;
BEGIN
  objs := obj^.sphereptr;
  veksub(hpoint,objs^.center,normal);
  veknormalize(normal);
END;
```

Abb. 19.13 Kugeltexturen nach dem Raytracing

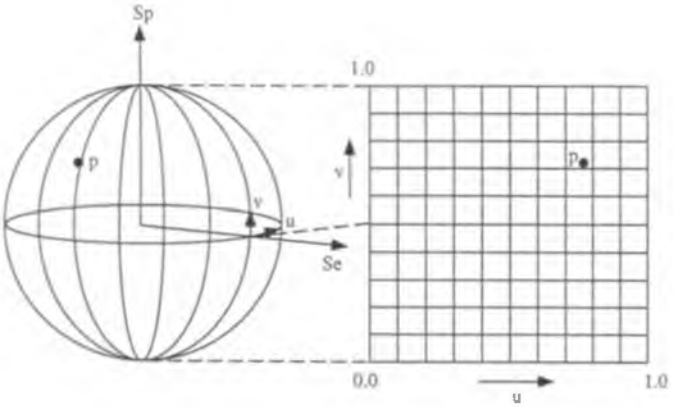


Abb. 19.14 Die Abbildung einer Kugelfläche auf ein Textur-Rechteck

19.2.1.2 Texturen und Kugeln

Soll eine Kugel mit einem Muster versehen werden, so muß der Shader eine Transformation von den Koordinaten des Musters auf die Kugelkoordinaten durchführen. Abb. 19.13 zeigt ein Beispiel für eine Textur auf einer Kugel. Als Information liegt der Schnittpunkt der Kugel sowie eine Textur in einem quadratischen Feld vor. Den Schnittpunkt muß man nun in einen Längen- und Breitengrad umrechnen, die man dann als Index in die Textur verwenden kann. Es gibt natürlich auch andere Methoden der Abbildung. In jedem Fall entstehen natürlich Verzerrungen, die man bei der Textur entsprechend berücksichtigen muß. An zusätzlicher Information wird noch ein Orientierungsvektor in der Kugel benötigt, um so zu erfahren, wo Nord- und Südpol liegt. Abb. 19.14 zeigt den Vorgang bei der Abbildung. Die Vektoren Sp (Pol) und Se (Äquator) bestimmen die Orientierung der Kugel. Der Punkt p soll dann auf das Textur-Gitter abgebildet werden. Dazu können folgende Formeln verwendet werden

$$\Phi = \text{acos}(-(N \cdot Sp))$$

$$
\begin{aligned}
v &= \frac{\Phi}{\pi} \\
\phi &= \frac{\operatorname{acos}\left(\frac{(Se \cdot N)}{\sin \Phi}\right)}{2\pi} \\
u &= \begin{cases} \phi & ((Sp \times Se) \cdot N) > 0 \\ 1 - \phi & ((Sp \times Se) \cdot N) \leq 0 \end{cases}
\end{aligned}
$$

u und v liegen dabei zwischen 0 und 1. Sie ergeben sich aus dem Breiten- bzw. Längengrad Φ und ϕ. N ist die Normale im Schnittpunkt zwischen Kugel und Sichtstrahl. Das Kreuz-Produkt $Sp \times Se$ kann beim Anlegen der Datenstruktur für die Kugel im voraus berechnet werden, und so spart man sich Zeit beim Raytracing-Vorgang. Wichtig ist, daß Se und Sp senkrecht aufeinander stehen. Dies wird vom Algorithmus vorausgesetzt. Die Abfrage am Schluß prüft nur, auf welcher Seite des Se-Vektors der Schnittpunkt liegt.

19.2.2 Quadriken

Um Objekte wie Zylinder, Kegel, Ellipsoide usw. erfassen zu können, gibt es die Möglichkeit, den vorherigen Algorithmus für die Kugel zu erweitern. Alle diese Objekte gehören nämlich zur Gruppe der Quadriken, auch als Flächen zweiter Ordnung bezeichnet. Die allgemeine Gleichung zur Beschreibung eines solchen Objekts lautet:

$$
\begin{pmatrix} x & y & z & 1 \end{pmatrix} \cdot \begin{pmatrix} A & B & C & D \\ B & E & F & G \\ C & F & H & I \\ D & G & I & J \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0
$$

Ausmultipliziert lautet die Gleichung:

$$
Ax^2 + 2Bxy + 2Cxz + 2Dx + Ey^2 + 2Fyz + 2Gy + Hz^2 + 2Iz + J = 0.
$$

Die Gleichung läßt sich nach dem Einsetzen der Geradengleichung ebenfalls mit der binomischen Formel lösen. Es ergibt sich dabei:

$$
\begin{aligned}
A_q &= Ax_d^2 + 2Bx_dy_d + 2Cx_dz_d + Ey_d^2 + 2Fy_dz_d + Hz_d^2 \\
B_q &= 2(Ax_0x_d + B(x_0y_d + x_dy_0) + C(x_0z_d + x_dz_0) \\
&\quad + Dx_d + Ey_0y_d + F(y_0z_d + y_dz_0) \\
&\quad + Gy_d + Hz_0z_d + Iz_d) \\
C_q &= Ax_0^2 + 2Bx_0y_0 + 2Cx_0z_0 + 2Dx_0 + Ey_0^2 \\
&\quad + 2Fy_0z_0 + 2Gy_0 + Hz_0^2 + 2Iz_0 + J
\end{aligned}
$$

Mit diesen Abkürzungen kann man nun die binomische Formel besser erkennen, die aber nur ausgewertet werden muß, wenn $B_q^2 - 4A_qC_q \geq 0$ gilt.

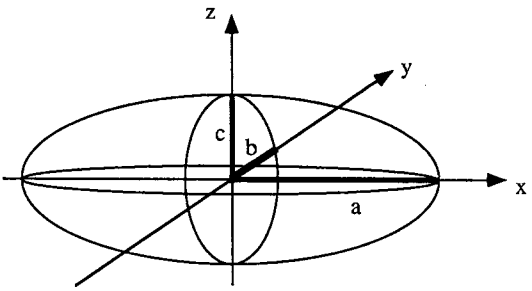


Abb. 19.15 Ellipsoid mit eingezeichneten Halbachsen

$$t_1 = \frac{-B_q - \sqrt{B_q^2 - 4A_qC_q}}{2A_q}$$
$$t_2 = \frac{-B_q + \sqrt{B_q^2 - 4A_qC_q}}{2A_q}$$

Wenn $A_q = 0$ ist, dann gibt es nur einen Schnittpunkt und es gilt:

$$t = -\frac{C_q}{B_q}.$$

Nun fehlt noch die Berechnung des Normalenvektors. Der Schnittpunkt selbst läßt sich durch Einsetzen in die Geradengleichung ermitteln. Der Normalenvektor wird aus dem Schnittpunkt $\begin{pmatrix} x & y & z \end{pmatrix}$ wie folgt bestimmt:

$$\begin{aligned} X_n &= Ax + By + Cz + D \\ Y_n &= Bx + Ey + Fz + G \\ Z_n &= Cx + Fy + Hz + I. \end{aligned}$$

Der Vektor $N = \begin{pmatrix} X_n & Y_n & Z_n \end{pmatrix}$ muß dann noch normalisiert werden, bevor er für das Raytracing weiter verwendet werden kann. Nun bleibt nur noch, für verschiedene Objekte die Koeffizienten A bis J zu bestimmen. Dies geschieht am einfachsten durch Koeffizientenvergleich mit der gewünschten Gleichung. Dazu folgen einige Beispiele in den nächsten Abschnitten.

19.2.2.1 Ellipsoid

Eine Kugel ist als Sonderfall des Ellipsoids beschreibbar. Das Ellipsoid hat die allgemeine Form:

$$\frac{(x - x_m)^2}{a^2} + \frac{(y - y_m)^2}{b^2} + \frac{(z - z_m)^2}{c^2} = 1$$

Dabei sind a, b und c die Halbachsen des Ellipsoids und x_m, y_m und z_m geben den Mittelpunkt an. Abb. 19.15 zeigt das Ellipsoid mit den eingetragenen Halbachsen. Die dazugehörige ausmultiplizierte Formel sieht dann so aus:

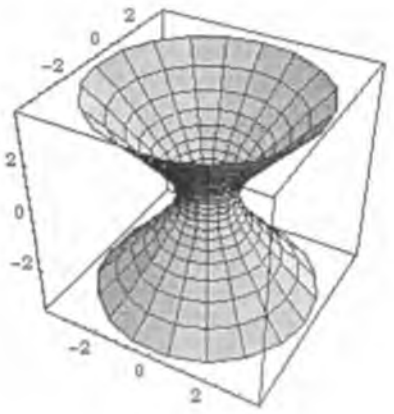


Abb. 19.16 Das Hyperboloid

$$\frac{x^2}{a^2} - \frac{2x_m}{a^2}x + \frac{y^2}{b^2} - \frac{2y_m}{b^2}y + \frac{z^2}{c^2} - \frac{2z_m}{c^2}z + \left(\frac{x_m^2}{a^2} + \frac{y_m^2}{b^2} + \frac{z_m^2}{c^2} - 1\right) = 0$$

Danach erhält man durch Koeffizientenvergleich die Werte für A bis J, in diesem Fall:

$$\begin{array}{rcl@{\quad}rcl@{\quad}rcl} A & = & \frac{1}{a^2} & E & = & \frac{1}{b^2} & H & = & \frac{1}{c^2} \\ B = C & = & 0 & F & = & 0 & I & = & -\frac{z_m}{c^2} \\ D & = & -\frac{x_m}{a^2} & G & = & -\frac{y_m}{b^2} & J & = & \left(\frac{x_m^2}{a^2} + \frac{y_m^2}{b^2} + \frac{z_m^2}{c^2} - 1\right) \end{array}$$

19.2.2.2 Hyperboloid

Durch Modifikation der Gleichungen kann man Hyperboloide darstellen. Abb. 19.16 zeigt ein einschaliges Hyperboloid. Die Formel dazu lautet:

$$\frac{(x - x_m)^2}{a^2} + \frac{(y - y_m)^2}{b^2} - \frac{(z - z_m)^2}{c^2} = 1$$

Dieser Körper ist nach oben und unten unbegrenzt. Wenn man ihn abschließen will, so muß der Raytracer ergänzt werden. Wie man so etwas tun kann, zeigen wir noch im Abschnitt zur Implementierung bei einem Kegelstumpf. Eine andere Möglichkeit ist es, CSG zu verwenden, z.B. bei einer UND-Verknüpfung zwischen Kugel und Hyperboloid.

Neben der einschaligen Version des Hyperboloids gibt es auch eine zweischalige. Dann gilt die folgende Formel:

$$\frac{(x - x_m)^2}{a^2} + \frac{(y - y_m)^2}{b^2} - \frac{(z - z_m)^2}{c^2} = -1$$

Wenn Sie die Formeln ausmultiplizieren, so können die Koeffizienten A bis J bestimmt werden, wie schon zuvor gezeigt.

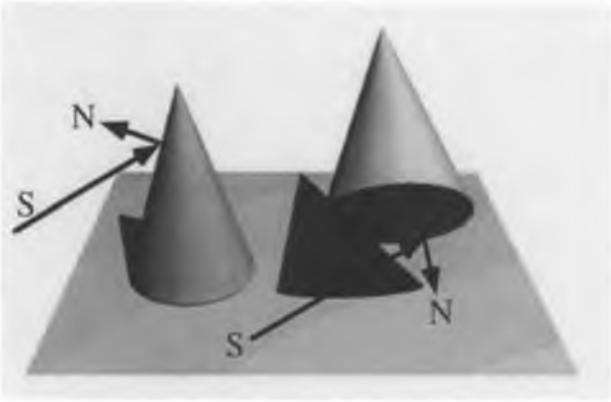


Abb. 19.17 Der Kegel im Raytracer

19.2.2.3 Kegel

Der Kegel ist eine Sonderform des Hyperboloids und die Formel dazu lautet:

$$\frac{(x-x_m)^2}{a^2}+\frac{(y-y_m)^2}{b^2}-\frac{(z-z_m)^2}{c^2}=0$$

Der Kegel ist in dieser Form ein unendlich großer Doppelkegel und man muß beim Programm zusätzliche Maßnahmen ergreifen, um wirklich nur einen Kegel darzustellen, z.B. mit Hilfe von CSG. Eine Besonderheit dieses Kegels ist die Angabe der beiden Halbachsen a und b. Damit erhält man einen elliptischen Querschnitt. Abb. 19.17 zeigt einen Kegel mit eingezeichneten Schnittstrahlen. Will man den Boden darstellen, so muß man dies im Raytrace-Algorithmus getrennt berücksichtigen. Das gleiche gilt auch für die Spitze des Kegelstumpfs, denn nur mit der Gleichung würde man einen Doppelkegel sehen, der sich unendlich nach oben und unten ausdehnt.

Die Abbildung einer Textur auf einen Kegel bzw. Kegelstumpf ist relativ leicht durchzuführen und Abb. 19.18 zeigt den so schattierten Kegel. Die Formel gilt für einen symmetrischen Kegelstumpf, also für $a = b$. Der Kegel besitzt die Höhe h mit einem Radius r_0 bei $z = 0$ und r_h bei $z = h$. Die Gleichung für den Kegel, der hier auf der xy-Ebene steht, lautet somit:

$$x^2+y^2=(r_0+(r_h-r_0)\cdot\frac{z}{h})^2$$

Die Abbildung auf die uv-Ebene errechnet sich dann zu:

$$\begin{aligned} v &= \frac{z}{h} \\ u' &= \frac{\operatorname{acos}\left(\frac{x}{r_0+(r_h-r_0)\cdot z/h}\right)}{2\pi} \\ u &= \begin{cases} 1-u' & y<0 \\ u' & y\geq 0 \end{cases} \end{aligned}$$

Abb. 19.18 Der Kegel und Texturen

Auch diese Formel gilt nur für einen stehenden Kegelstumpf. In diese Formel kann auch leicht eine Prüfung für den Raytracer eingebaut werden. Für $v < 0$ oder $v > 1$ wird der Kegelstumpf nicht getroffen. Dann liefert die Schnittroutine einfach den Wert `FALSE` und schon sieht man auf dem Bildschirm nur den Kegelstumpf. Auf ähnliche Weise wird im späteren Abschnitt zur Implementierung für einen Kegelstumpf in unserem Raytracer auch tatsächlich verfahren. Der Kegelstumpf erscheint dann allerdings hohl, man sieht keine Deckel.

19.2.2.4 Elliptischer Zylinder

Der Zylinder ist wiederum eine Sonderform des Hyperboloids: der Term für die z-Koordinate entfällt.

$$\frac{(x - x_m)^2}{a^2} + \frac{(y - y_m)^2}{b^2} = 1$$

Dieser Zylinder steht senkrecht auf der xy-Ebene. Will man eine andere Orientierung, so muß man, wie bei allen Grundkörpern, eine Koordinatentransformation durchführen. Abb. 19.19 zeigt einen Zylinder mit eingezeichneten Vektoren. Dabei ist zu beachten, daß der Zylinder in der allgemeinen Form unendlich lang ist und auch beim Raytracen keinen Boden oder Deckel zeigen wird. Dies ist auch wieder gesondert zu behandeln. Eine Möglichkeit wäre, die Texturtransformation zu verwenden, um so den abzubildenden Bereich festzustellen. In Abb. 19.20 ist ein Zylinder mit Textur dargestellt. Die Formel dazu ist recht schlicht und lautet:

$$\begin{aligned} v &= \frac{z}{h} \\ u' &= \frac{\operatorname{acos}(\frac{x}{r})}{2\pi} \\ u &= \begin{cases} 1 - u' & y < 0 \\ u' & y \geq 0 \end{cases} \end{aligned}$$

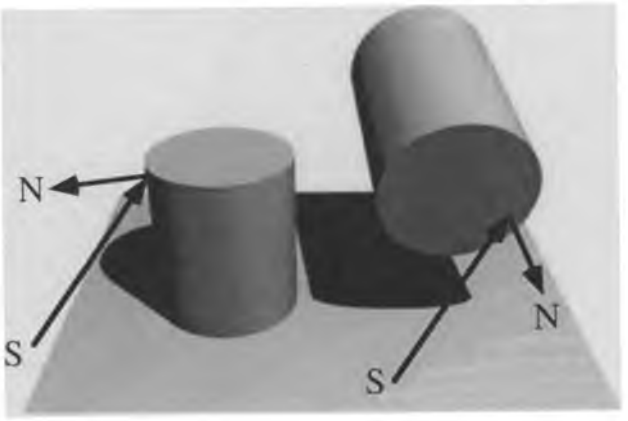


Abb. 19.19 Der Zylinder im Raytracer

Abb. 19.20 Der Zylinder und Texturen

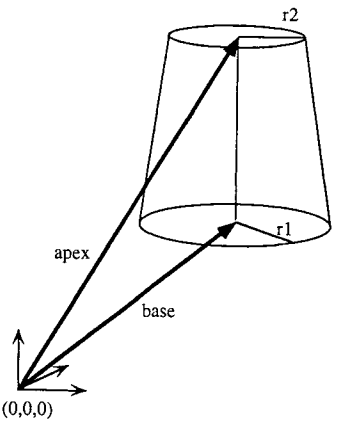


Abb. 19.21 Der symmetrische Kegelstumpf und die Definition für den Raytracer

Dabei ist h die Höhe des Zylinders und r der Radius. Wenn $v < 0$ oder $v > 1$, dann wird die Schnittroutine den Wert `FALSE` zurückliefern. Der Zylinder erscheint dann als Hohlkörper auf dem Bildschirm.

Im nächsten Abschnitt werden wir die im Raytracer verwendete Routine kennenlernen, die Kegel, Kegelstumpf und Zylinder beliebig im Raum stehend bearbeiten kann.

19.2.2.5 Implementierung für den symmetrischen Kegelstumpf

Hier wollen wir eine Möglichkeit für die Darstellung eines Kegelstumpfs aufzeigen. Dabei soll der Kegelstumpf wahlweise mit oder ohne Deckel dargestellt werden können. Der Kegelstumpf darf beliebig im Raum stehen. Abb. 19.21 zeigt die Vektoren zur Definition des Kegelstumpfs. `base` zeigt dabei auf die Mitte der Unterseite und `apex` auf die Mitte der Oberseite. Ferner wird der Radius r_1 für `base` und der Radius r_2 für `apex` als Parameter benötigt. Der Kegelstumpf kann damit beliebig im Raum positioniert werden und die Orientierung ist frei wählbar.

An dieser Stelle wollen wir uns einmal ein Einleseprogramm näher ansehen, das diese Parameter verarbeitet und weitere, zur Berechnung der Schnittpunkte nötige Parameter ermittelt und festhält. Dabei werden die Daten in einer Struktur mit folgendem Aussehen festgehalten:

```
conerec = RECORD
  cone_base : vektor;          (**)
  cone_base_radius : DOUBLE;   (**)
  cone_base_d : DOUBLE;
  cone_apex : vektor;          (**)
  cone_apex_radius : DOUBLE;   (**)
  cone_apex_d : DOUBLE;
  cone_u : vektor;
  cone_v : vektor;
  cone_w : vektor;
```

```
  cone_height : DOUBLE;
  cone_slope : DOUBLE;
  cone_min_d : DOUBLE;
  cone_max_d : DOUBLE;
  cone_solid : BOOLEAN;        (**)
     (* TRUE = mit Deckel *)
END;
```

Nur die durch (**) markierten Informationen werden vom Benutzer vorgeben. Alle anderen Parameter werden wie folgt berechnet:

```
PROCEDURE enter_cone(basepoint : vektor; baseradius : DOUBLE;
                     apexpoint : vektor; apexradius : DOUBLE;
                     solid : BOOLEAN;
                     surfptr : surfpp);
```

Als Parameter wird zusätzlich ein Zeiger auf die Oberflächeneigenschaften übergeben (surfpp), der aber für die Schnittpunktberechnung ignoriert werden kann und erst für das Shading relevant ist.

```
VAR
  spptr            : conepp;
  ppptr            : primpp;
  dmin,dmax,d,ftmp : DOUBLE;
  tmpv             : vektor;
  i                : INTEGER;
BEGIN
```

An dieser Stelle werden die Datenstrukturen für den Konus angelegt.

```
  new(worldptr[worldstack]); (* neuer Welt-Pointer anlegen *)
  worldptr[worldstack]^.objectart := cone; (* Objekt Typ *)
  new(worldptr[worldstack]^.coneptr);      (* Konus Info *)
  spptr := worldptr[worldstack]^.coneptr;  (* und merken *)
  ppptr := worldptr[worldstack];  (* fuer schnellen Zugriff *)
```

Damit sind zwei Pointer definiert, die im weiteren Verlauf verwendet werden, um die Datenstrukturen zur vervollständigen. Es werden erst einmal alle bekannten Daten gespeichert.

```
  IF baseradius = 0 THEN baseradius := 1; (* Fehlerschutz *)
  IF apexradius = 0 THEN apexradius := 1;
  spptr^.cone_base := basepoint;
  spptr^.cone_apex := apexpoint;
  spptr^.cone_base_radius := baseradius;
  spptr^.cone_apex_radius := apexradius;
```

cone_w ist der Vektor zwischen Basis und Oberseite des Konus. Er wird hier ebenfalls schon vorausberechnet, da er bei der Schnittbestimmung benötigt wird. Ferner läßt sich daraus ein Normalenvektor für die Deckel bestimmen.

```
  veksub(apexpoint,basepoint,spptr^.cone_w);
```

Die Höhe des Konus läßt sich ebenfalls daraus ableiten, die Funktion `fvektnormalize` liefert die Höhe als Ergebnis zurück und normalisiert den als Parameter übergebenen Vektor.

```
spptr^.cone_height := fveknormalize(spptr^.cone_w);
```

Als nächstes wird die Steigung des Konus bestimmt.

```
spptr^.cone_slope :=
   (spptr^.cone_apex_radius - spptr^.cone_base_radius)
                  / spptr^.cone_height;
```

Zwei Skalarprodukte sind für die Deckel-Berechnung wichtig und die Basis wird zusätzlich für den Normalenvektor der Außenseite benötigt.

```
spptr^.cone_base_d := -vekdot(basepoint,spptr^.cone_w);
spptr^.cone_apex_d := -vekdot(apexpoint,spptr^.cone_w);
```

Hier wird nur das übergebene Flag `solid` festgehalten. Ist es `TRUE`, dann wird der Konus als fester Körper (mit Deckel) dargestellt.

```
spptr^.cone_solid := solid;  (* merken *)
```

Für die Schnittberechnung werden anschließend weitere Parameter berechnet, `cone_u` und `cone_v`:

```
vekset(0.0,0.0,1.0,tmpv);
IF ((1.0 - abs(vekdot(tmpv,spptr^.cone_w)))  < rayeps) THEN BEGIN
  vekset(0.0,1.0,0.0,tmpv);
END;
vekcross(spptr^.cone_w,tmpv,spptr^.cone_u);
vekcross(spptr^.cone_u,spptr^.cone_w,spptr^.cone_v);
veknormalize(spptr^.cone_u);
veknormalize(spptr^.cone_v);
```

Für die Prüfung, ob der Strahl außerhalb des Kegelstumpfes liegt, werden zwei Werte `cone_min_d` und `cone_max_d` berechnet. Dies sind die Abstände der Deckel zum Koordinatenursprung. Damit ist bei der Schnittroutine eine einfache Abfrage möglich.

```
spptr^.cone_min_d := vekdot(spptr^.cone_w,spptr^.cone_base);
spptr^.cone_max_d := vekdot(spptr^.cone_w,spptr^.cone_apex);
IF (spptr^.cone_max_d < spptr^.cone_min_d) THEN BEGIN
  ftmp := spptr^.cone_max_d;
  spptr^.cone_max_d := spptr^.cone_min_d;
  spptr^.cone_min_d := ftmp;
END;
```

Nun werden nur noch Pointer definiert und allgemeine Verwaltungen durchgeführt.

```
    ppptr^.csgop := csgprim;
    ppptr^.surfptr := surfptr;
    ppptr^.left := NIL;
    ppptr^.right := NIL;
    worldstack := worldstack + 1;
    IF worldstack > NESTOPS THEN BEGIN
      closegraph;
      writeln('Fehler: worldstack ueberlauf');
    END;
  END;
```

Wenn danach der Raytracer aktiv wird, muß die Schnittroutine aufgerufen werden. Die Schnittroutine für den Konus sieht dann so aus:

```
FUNCTION inter_cone(obj : primpp; ray : rayrec;
                    VAR hitdata:isectrec) : BOOLEAN;
```

Als Parameter wird die Objekt-Datenstruktur und der Sichtstrahl übergeben. Als Ergebnis liefert die Routine `TRUE`, wenn ein Schnitt erfolgte. Dann stehen in `hitdata` alle für den Shader benötigte Informationen.

```
VAR
  tray : rayrec;
  objs : conepp;
  v,p : vektor;
  a,b,c,d,disc : DOUBLE;
  tcheck,t1,t2 : DOUBLE;
  t1deckel,t2deckel : DOUBLE;
  flag,flag1,flag2 : BOOLEAN;  (* Deckel hit flags *)
  nroots : DOUBLE;
BEGIN
  objs := obj^.coneptr;
```

Zunächst erfolgt eine Koordinatentransformation. Der Beobachterpunkt und der Richtungsvektor werden in das lokale Koordinatensystem umgerechnet (`tray.point` und `tray.direction`).

```
  veksub(ray.point,objs^.cone_base,v);
  tray.point.x := vekdot(v,objs^.cone_u);
  tray.point.y := vekdot(v,objs^.cone_v);
  tray.point.z := vekdot(v,objs^.cone_w);
  tray.direction.x := vekdot(ray.direction,objs^.cone_u);
  tray.direction.y := vekdot(ray.direction,objs^.cone_v);
  tray.direction.z := vekdot(ray.direction,objs^.cone_w);
```

Damit ist die Schnittberechnung unabhängig von der Lage des Konus und nun kann die eigentliche Schnittberechnung so erfolgen, wie wir sie für Quadriken gezeigt haben. Es werden zunächst die Parameter `a,b` und `c` berechnet, um sie danach in die binomische Formel einsetzen zu können.

```
a := tray.direction.x * tray.direction.x
     + tray.direction.y * tray.direction.y
    - objs^.cone_slope * objs^.cone_slope *
      tray.direction.z * tray.direction.z;
b := 2.0 * (tray.point.x * tray.direction.x +
           tray.point.y * tray.direction.y
           - objs^.cone_slope * objs^.cone_slope
           * tray.point.z * tray.direction.z
           - objs^.cone_base_radius * objs^.cone_slope
           * tray.direction.z);
c := objs^.cone_slope * tray.point.z + objs^.cone_base_radius;
c := tray.point.x * tray.point.x  + tray.point.y * tray.point.y
    - c*c;
```

Die Prüfung, ob überhaupt ein Schnitt mit dem Konus erfolgt, wird durch Prüfen des Wertes `disc` durchgeführt. Dabei ist der Kegel noch unendlich lang. Dennoch kann der Sichtstrahl daran vorbeilaufen.

```
disc := b * b - 4.0 * a * c;
```

Wenn `disc` < 0, kann man die Routine schon hier verlassen, denn auch ein Schnitt mit den Deckeln ist nicht mehr möglich.

```
IF (disc < 0.0) THEN BEGIN
  flag := FALSE;  (* sonst trifft Strahl ueberhaupt nicht *)
END ELSE BEGIN (* weitere Untersuchung *)
```

Ansonsten muß weiter untersucht werden.

```
  hitdata.usenormal := FALSE;  (* Default setzen *)
  hitdata.use2normal := FALSE;
  flag1 := FALSE;
  flag2 := FALSE;
```

Es erfolgt hier zunächst der Check, ob der Strahl den Deckel schneidet. Jedoch nur, wenn das Flag `solid=TRUE` ist, also der Konus auch einen Deckel besitzt. Dazu wird das Unterprogramm `hit_plane` verwendet. Es prüft den Schnitt zwischen Ebene und Sichtstrahl. Dazu wird als Parameter der Normalenvektor der Ebene (in `cone_w`), und das Skalarprodukt zwischen Basisvektor und Normale (in `cone_base_d` bzw. `cone_apex_d`) sowie der Sichtstrahl übergeben. Als Ergebnis wird der Parameter `t` für den Schnittpunkt mit der Ebene berechnet.

```
  IF objs^.cone_solid THEN BEGIN
    IF (hit_plane(objs^.cone_w,objs^.cone_base_d,
          ray,t1deckel)) THEN BEGIN
      IF (hit_plane(objs^.cone_w,objs^.cone_apex_d,
            ray,t2deckel)) THEN BEGIN
```

Für beide Schnittpunkte wird danach geprüft, ob sie innerhalb des Radius für Basis oder Oberseite liegen. `flag1` bzw. `flag2` merken sich, ob die Deckel getroffen wurden. Die Routine `hit_disc` übernimmt diese Prüfung (siehe auch Abschnitt 19.2.3.2).

```
      IF (t1deckel >= rayeps) THEN BEGIN
        flag1 := hit_disc(objs^.cone_base,
                    objs^.cone_base_radius, ray,t1deckel);
      END;
      IF (t2deckel >= rayeps) THEN BEGIN
        flag2 := hit_disc(objs^.cone_apex,
                    objs^.cone_apex_radius, ray,t2deckel);
      END;
    END; (* hit 2 *)
  END; (* hit 1 *)
END; (* solid check *)
```

Dann wird die binomische Formel weiter ausgerechnet.

```
disc := sqrt(disc);
```

Die folgende Zeile verhindert einen entarteten Kegel.

```
IF a = 0 THEN a:=0.002;
```

Man erhält nun normalerweise zwei Schnittpunkte, es sei denn, der Strahl schneidet tangential, was aber für den Verlauf des Algorithmus nicht von Bedeutung ist.

```
t1 := (-b - disc) / (2.0 * a);
t2 := (-b + disc) / (2.0 * a);
IF (t2 < rayeps) AND (t1 < rayeps) THEN BEGIN
```

Wenn beide Schnittpunkte hinter dem Beobachter liegen, so ist das Objekt auch nicht sichtbar und man kann die Berechnungen beenden.

```
  flag := FALSE;
END ELSE BEGIN (* weitere Untersuchung sonst *)
  IF t1 < rayeps THEN BEGIN
```

Der Beobachter ist im Inneren des Konus, daher ist nur ein Schnittpunkt gültig und der wird in `tcheck` festgehalten.

```
    nroots := 1; (* t2 gueltig *)
    tcheck := t2;  t2 := t1; (* exchange *)
  END ELSE IF t2 < rayeps THEN BEGIN
```

Hier ist ebenfalls nur ein Schnittpunkt gültig.

```
    nroots := 1;
    tcheck := t1; (* t2 remains *)
  END ELSE BEGIN
```

Der Beobachter befindet sich außerhalb des Kegelstumpfes und damit müssen beide Schnittpunkte weiterverarbeitet werden.

```
    nroots := 2;
  END;
```

Es wird als nächstes geprüft, ob der Schnittpunkt, der mit dem unendlich langen Konus berechnet wurde, auch innerhalb des Bereichs für den Kegelstumpf liegt. Zunächst für den Fall, daß der Betrachter innerhalb des Kegelstumpfs liegt, was z.B. bei Brechungen auftreten kann.

```
IF nroots = 1 THEN BEGIN (* innerhalb des Kegels? *)
  raypoint(ray,tcheck,p);  (* t2 ist der erste Punkt *)
  d := vekdot(objs^.cone_w,p);
  IF (d >= objs^.cone_min_d)
      AND (d <= objs^.cone_max_d) THEN BEGIN
```

Der Schnittpunkt liegt innerhalb des Kegelstumpfes und wird gespeichert.

```
    hitdata.t := tcheck;
    IF objs^.cone_solid THEN BEGIN
```

Wenn die Deckel verwendet werden, muß für CSG auch der zweite Schnittpunkt hinter dem Beobachter festgehalten werden, sonst ist der Körper unendlich dünn und kann bei CSG nicht als Subtrahend verwendet werden.

```
      hitdata.t2 := t2;  (* zweiten Punkt, wenn solid *)
    END ELSE BEGIN
```

Dann fallen beide Schnittpunkte zusammen.

```
      hitdata.t2 := tcheck;
    END;
    IF objs^.cone_solid THEN BEGIN
```

Bei CSG mit Deckel wird `enterflag` als `FALSE` festgehalten, denn der Strahl tritt aus dem Körper aus.

```
      hitdata.enterflag := FALSE;  (* liegt im Inneren *)
      hitdata.enter2flag := FALSE;
    END ELSE BEGIN
```

Bei einem hohlen Kegelstumpf hingegen gibt es kein Inneres, sondern nur eine dünne Außenschale.

```
      hitdata.enterflag := TRUE;
      hitdata.enter2flag := TRUE;
    END;
```

In der Datenstruktur wird das getroffene Objekt festgehalten und später bei der Schnittauswertung und beim Shader weiter verwendet.

```
    hitdata.primptr := obj; (* hier immer das gleiche *)
    hitdata.prim2ptr := obj;
    flag := TRUE;
  END ELSE BEGIN (* war nichts *)
    flag := FALSE;
  END;
END ELSE BEGIN (* 2 Wurzeln gefunden *)
```

An dieser Stelle befindet sich der Beobachter außerhalb des Kegelstumpfs. Es muß nun bestimmt werden, ob die Deckel oder nur die Außenwände getroffen werden. Dazu wird zunächst einmal berechnet, ob die Schnitte überhaupt den Kegelstumpf getroffen haben. `t1` ist hier automatisch der näher liegende Punkt.

```
raypoint(ray,t1,p); (* hier t1 naeher *)
d := vekdot(objs^.cone_w,p);
IF ( d >= objs^.cone_min_d)
     AND (d <= objs^.cone_max_d) THEN BEGIN
```

Daher wird zunächst `t1` untersucht.

```
  hitdata.t := t1;                    (* erster Punkt ok *)
  IF objs^.cone_solid THEN BEGIN
    hitdata.t2 := t2;      (* zweiten Punkt, wenn solid *)
  END ELSE BEGIN
    hitdata.t2 := t1;       (* zweiter Punkt weiter weg *)
  END;
  hitdata.enterflag := TRUE;  (* von Aussen nach Innen *)
  hitdata.enter2flag := TRUE;
  hitdata.primptr := obj;
  hitdata.prim2ptr := obj;
  flag := TRUE;
END ELSE BEGIN      (* vielleicht dann zweiter Punkt ok *)
```

Wenn der erste Punkt `t1` außerhalb liegt, dann könnte der zweite Punkt `t2` den Kegelstumpf treffen.

```
raypoint(ray,t2,p);
d := vekdot(objs^.cone_w,p);
IF ( d >= objs^.cone_min_d)
   AND (d <= objs^.cone_max_d) THEN BEGIN
```

Diesmal war es der zweite Schnittpunkt `t2`.

```
  hitdata.t := t2;                  (* zweiter Punkt ok *)
  IF objs^.cone_solid THEN BEGIN
    hitdata.t2 := t1;      (* ersten Punkt, wenn solid *)
  END ELSE BEGIN
    hitdata.t2 := t2;       (* erster Punkt weiter weg *)
  END;
  IF objs^.cone_solid THEN BEGIN
    hitdata.enterflag := FALSE;          (* liegt Innen *)
    hitdata.enter2flag := FALSE;
  END ELSE BEGIN
    hitdata.enterflag := TRUE;  (* liegt immer Aussen *)
    hitdata.enter2flag := TRUE;
  END;
  hitdata.primptr := obj;
  hitdata.prim2ptr := obj;
  flag := TRUE;
```

```
        END ELSE BEGIN                          (* auch nicht ok *)
          flag := FALSE;
        END;
      END;
    END;
  END; (* t2 check *)
END; (* disc check *)
```

Wenn der Kegelstumpf massiv ist und als Festkörper dargestellt werden soll, dann muß nun untersucht werden, welche der möglichen vier Schnitte tatsächlich verwendet werden sollen.

```
IF objs^.cone_solid THEN BEGIN (* nur wenn solid definiert *)
  IF flag1 OR flag2 THEN BEGIN (* trifft auch Deckel *)
    IF flag THEN BEGIN         (* schneidet auch Konus *)
      IF flag2 THEN BEGIN      (* es kann nur einer beider *)
        t1deckel := t2deckel;  (* Deckel getroffen werden *)
      END;
      (* Auch nur von Aussen nach Innen moeglich *)
      (* sonst wuerde entweder cone FALSE, oder Deckel *)
      IF t1deckel < hitdata.t THEN BEGIN (* Deckel im Weg *)
```

Der Deckel liegt näher und ist daher sichtbar. In `hitdata.t2` wird daher der zweite Schnitt gespeichert.

```
        hitdata.t2 := hitdata.t;  (* bildet zweiten Treffer *)
        hitdata.t := t1deckel;
        hitdata.usenormal := TRUE;
        hitdata.normalvec := objs^.cone_w;
      END;
    END ELSE BEGIN (* trifft nur den Deckel *)
```

Hier werden nur die Deckel getroffen.

```
      IF flag1 AND flag2 THEN BEGIN (* beide Seiten *)
        hitdata.usenormal := TRUE;
        hitdata.use2normal := TRUE;
        hitdata.normalvec := objs^.cone_w;
        hitdata.normal2vec := objs^.cone_w;
        t1 := t1deckel;
        t2 := t2deckel;
```

Es muß nun noch bestimmt werden, welcher der beiden Deckel näher beim Betrachter liegt.

```
        IF (t2 < t1) THEN BEGIN
          t2 := t1deckel;
          t1 := t2deckel;
        END;
        hitdata.t := t1; (* erster Punkt hier wieder t2 *)
        hitdata.t2 := t2;  (* zweiter Punkt weiter weg *)
```

```
            hitdata.enterflag := TRUE;  (* liegt aussen *)
            hitdata.enter2flag := TRUE;
            hitdata.primptr := obj; (* hier immer das gleiche *)
            hitdata.prim2ptr := obj;
            flag := TRUE;
          END ELSE IF flag1 THEN BEGIN (* flag 1 only *)
```

Nur ein Deckel wird getroffen, da der Beobachter im Inneren sitzt.

```
            hitdata.usenormal := TRUE;
            hitdata.use2normal := TRUE;
            hitdata.normalvec := objs^.cone_w;
            hitdata.normal2vec := objs^.cone_w;
            hitdata.t := t1deckel; (* erster Punkt hier wieder t2 *)
            hitdata.t2 := t2deckel;  (* zweiter Punkt weiter weg *)
            hitdata.enterflag := FALSE;  (* liegt innen *)
            hitdata.enter2flag := FALSE;
            hitdata.primptr := obj; (* hier immer das gleiche *)
            hitdata.prim2ptr := obj;
            flag := TRUE;
          END ELSE BEGIN (* flag2 only *)
```

Ebenfalls nur ein Deckel, der Beobachter ist im Inneren.

```
            hitdata.usenormal := TRUE;
            hitdata.use2normal := TRUE;
            hitdata.normalvec := objs^.cone_w;
            hitdata.normal2vec := objs^.cone_w;
            hitdata.t := t2deckel; (* erster Punkt hier wieder t2 *)
            hitdata.t2 := t1deckel;  (* zweiter Punkt weiter weg *)
            hitdata.enterflag := FALSE;  (* liegt innen *)
            hitdata.enter2flag := FALSE;
            hitdata.primptr := obj; (* hier immer das gleiche *)
            hitdata.prim2ptr := obj;
            flag := TRUE;
          END;
        END; (* cone check *)
      END; (* flag1 or flag2 *)
    END; (* solid *)
    inter_cone := flag;
  END;
```

Damit ist die Schnittroutine beendet. Als nächstes muß die Normale bestimmt werden, jedoch nur dann, wenn der getroffene Konus auch das erste sichtbare Objekt war. In diesem Fall wird vom Raytracer die Normale angefragt und das geschieht mit der nachfolgenden Routine `normal_cone`.

```
  PROCEDURE normal_cone(obj : primpp; VAR hitdata : isectrec;
                        hpoint : vektor; VAR normal : vektor);
  VAR
```

```
    objs : conepp;
    t : DOUBLE;
    v : vektor;
  BEGIN
```

Als Parameter wird die Datenstruktur des Konus sowie der Schnittpunkt übergeben. Dann gibt es zwei Fälle. Wurden die Deckel getroffen, so wurde in der Datenstruktur `hitdata.usenormal` der Wert `TRUE` abgelegt (durch `inter_cone`).

```
    objs := obj^.coneptr;
    IF hitdata.usenormal THEN BEGIN (* Deckel getroffen *)
```

Dabei wird einfach der Normalenvektor der Deckel verwendet.

```
      normal := hitdata.normalvec; (* Schon berechnet *)
    END ELSE BEGIN
```

Sonst wurde die Außenhaut getroffen und die Normale muß in Abhängigkeit vom Schnittpunkt berechnet werden. Dazu dient auch das vorberechnete Vektorprodukt `cone_base_d`.

```
      t := - (vekdot(hpoint,objs^.cone_w) + objs^.cone_base_d);
      vekadds(t,objs^.cone_w,hpoint,v);
      veksub(v,objs^.cone_base,normal);
      veknormalize(normal);
      vekadds(-objs^.cone_slope,objs^.cone_w,normal,normal);
      veknormalize(normal);
    END;
  END;
```

Abb. XVII im farbigen Bildteil zeigt ein paar Kegelstümpfe als Hohlkörper in unterschiedlichen Positionen und Abb. XIX zeigt diese mit Deckel sowie einer Spiegelkugel zur genaueren Betrachtung.

19.2.3 Ebenen und flächige Objekte

Obwohl man Ebenen auch als Sonderfall der quadratischen Gleichung des vorherigen Abschnitts behandeln könnte, ist es effizienter, diese mit eigenen Algorithmen zu berechnen. Ferner möchte man eher flächige Objekte darstellen als ganze Ebenen. Der Beispiel-Raytracer auf der Diskette hat die Möglichkeit, Dreiecke zu verwenden, aus denen sich komplizierte Objekte leicht zusammensetzen lassen. Ferner haben wir noch eine Besonderheit für die Normalenberechnung vorgesehen, doch dazu später mehr.

19.2.3.1 Schnitt: Ebene und Strahl

Zunächst wollen wir einen Strahl mit einer Ebene schneiden. Die Ebene ist dabei durch einen Punkt auf der Ebene und einen Normalenvektor eindeutig bestimmt. Abb. 19.22 zeigt ein Schema dazu. Die Ebene ist durch den Vektor E_0 und die Normale E_n definiert. Der Sichtstrahl trifft an der Stelle t auf die

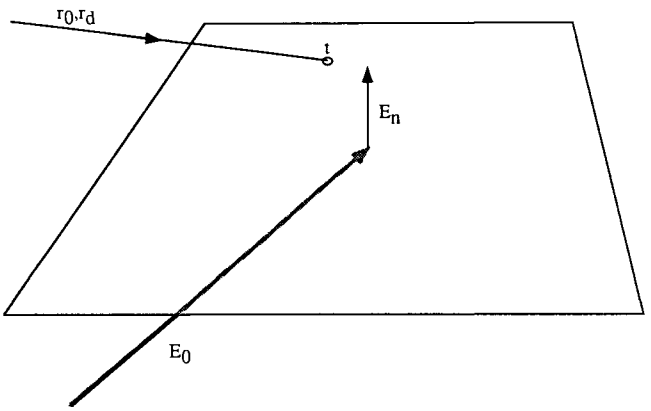


Abb. 19.22 Schnitt zwischen Strahl und Ebene

Ebene. Der Sichtstrahl ist durch den Vektor r_0 und r_d gegeben. Es gilt die allgemeine Ebenengleichung:

$$\left(\begin{pmatrix} A \\ B \\ C \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix}\right) + D = 0.$$

Dabei ist $\begin{pmatrix} A & B & C \end{pmatrix}^T$ der Normalenvektor der Ebene mit $A = x_n, B = y_n$ und $C = z_n$, und D der Abstand zum Koordinatenursprung. Dieser Abstand D läßt sich wie folgt berechnen (siehe auch Abschnitt 7.3):

$$D = \left(\begin{pmatrix} A \\ B \\ C \end{pmatrix} \cdot \begin{pmatrix} x_{E0} \\ y_{E0} \\ z_{E0} \end{pmatrix}\right).$$

Dabei ist $\begin{pmatrix} x_{E0} & y_{E0} & z_{E0} \end{pmatrix}^T$ ein Punkt auf dieser Ebene. Mit diesen Angaben ist es möglich, den Wert t zu bestimmen:

$$t = \frac{-((E_n \cdot r_0) + D)}{(E_n \cdot r_d)}$$

Zunächst muß der Nenner berechnet werden. Ist dieser Null, so verläuft der Sichtstrahl parallel zur Ebene (oder in dieser). Wenn nicht, so muß auch der Zähler berechnet werden und es gibt zwei Möglichkeiten. Wenn $t < 0$ ist, so wird die Ebene hinter dem Betrachter getroffen und ist nicht sichtbar. Nur Werte $t > 0$ werden daher genauer untersucht.

Zur Vertiefung ist hier der Algorithmus in Programmform abgedruckt.

```
FUNCTION hit_plane(normal : vektor; d : DOUBLE; ray: rayrec;
                   VAR t : DOUBLE):BOOLEAN;
VAR
  vd,v0 : DOUBLE;
BEGIN
```

Die Funktion `hit_plane` erhält als Parameter die Ebenen-Normale in `normal` und den Wert `d`, die Distanz zum Koordinatenursprung. In `ray` ist die Information des Sichtstrahls gespeichert. Als nächstes wird der Wert des Nenners berechnet.

```
  vd := vekdot(normal,ray.direction);
```

Wenn der Wert Null ist, so liefert die Funktion den Wert `FALSE`.

```
  IF vd = 0 THEN BEGIN
    hit_plane := FALSE;
    t := 0;
  END ELSE BEGIN
```

Sonst liegt ein Schnittpunkt vor.

```
    v0 := - (vekdot(normal,ray.point)+d);
```

In `t` steht der Schnittparameter. Den Schnittpunkt erhält man, wenn $r_0 + t \cdot r_d$ berechnet wird.

```
    t := v0 / vd;
```

Diese Routine liefert den Wert `t`, auch wenn er negativ ist, denn hier soll nur bestimmt werden, ob die Ebene geschnitten wird, nicht schon, ob sie auch sichtbar ist. Die Routine wird z.B. von `inter_cone` verwendet (siehe vorherigen Abschnitt).

```
    hit_plane := TRUE; (* auch wenn t < 0 *)
  END;
END;
```

19.2.3.2 Schnitt: Kreisfläche und Strahl

Interessanter, als Ebenen zu schneiden, ist die Verwendung flächiger Objekte auf der Ebene. Eine einfache Form ist die Kreisfläche. In unserem Raytracer wird sie verwendet, um die Deckel des Kegelstumpfs zu berechnen. In Abb. 19.23 sind die Parameter für die Kreisfläche nochmals dargestellt. Der Algorithmus setzt voraus, daß der Parameter t für den Schnittpunkt mit der dazugehörigen Ebene bereits berechnet wurde. Gegeben ist daher neben der Ebenengleichung zusätzlich der Radius der Kreisfläche.

```
FUNCTION hit_disc(base : vektor; r : DOUBLE;
                  ray : rayrec; t : DOUBLE) : BOOLEAN;
VAR
  p, p1: vektor;
BEGIN
```

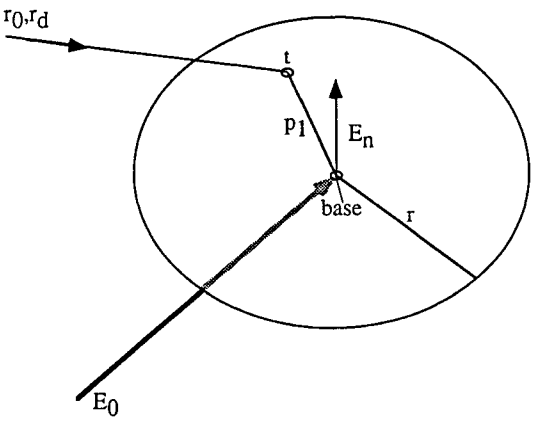


Abb. 19.23 Kreisscheiben im Raytracer

Es wird dann als erstes der Schnittpunkt selbst berechnet und im Vektor p abgelegt.

```
raypoint(ray,t,p);
```

Danach muß die Distanz zwischen dem Basispunkt der Kreisfläche (base) und dem Schnittpunkt bestimmt werden. Dazu wird in p1 der Vektor in der Ebene berechnet.

```
veksub(p,base,p1);  (* entlang radius *)
```

Die Länge des Vektors muß mit dem Radius der Scheibe verglichen werden. Ist sie kleiner, wird die Kreisfläche getroffen, sonst nicht.

```
IF (veklen(p1)) < r THEN
  hit_disc := TRUE ELSE hit_disc := FALSE;
END;
```

Auf ähnliche Weise könnte man den Algorithmus leicht für Ellipsenflächen umbauen. Dabei wird die Radiusabfrage entsprechend geändert und die Orientierung der Ellipse muß noch berücksichtigt werden.

19.2.3.3 Schnitt: konvexes Viereck und Strahl

Aus Vierecken kann man auch komplexe Formen konstruieren, und daher soll hier diese Grundform verwendet werden. Insbesondere existiert ein eleganter Algorithmus, der gleichzeitig die Textur-Koordinaten für ein konvexes Viereck berechnet. Damit läßt sich dann durch Bereichsprüfung einfach feststellen, ob der Schnittpunkt im Bereich des konvexen Vierecks liegt. Voraussetzung für diesen Test ist, daß das konvexe Viereck planar ist (also in einer Ebene liegt), nur dann arbeitet der Algorithmus korrekt. Zuvor kann man mit einem Ebenentest den Schnittpunkt bestimmen und genau da fangen wir mit dem Algorithmus an. Abb. 19.24 zeigt das dazugehörige Schema.

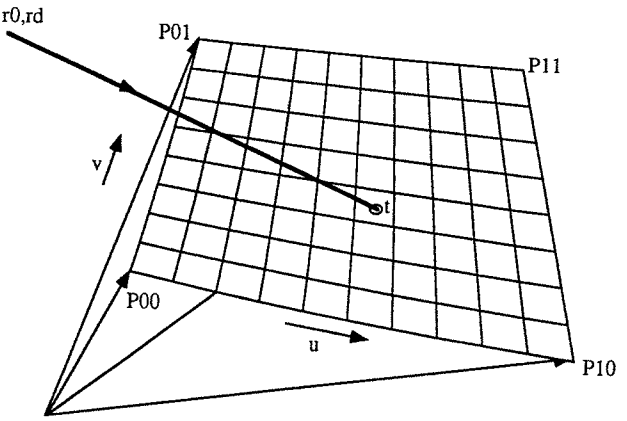


Abb. 19.24 Ein Viereck im Raytracer

Das Viereck ist durch vier Vektoren P_{00}, P_{01}, P_{10} und P_{11} definiert. Man muß nur darauf achten, daß es planar ist, da die Vektoren auch ein räumliches Gebilde aufspannen könnten.

Ist diese Voraussetzung gegeben, so kann man folgendermaßen vorgehen: Es werden die Texturkoordinaten u und v für den Schnittpunkt bestimmt. Liegen diese im Bereich $0 \ldots 1$, dann trifft der Sichtstrahl das Innere des Vierecks, sonst nicht. N ist dabei die Normale der aufgespannten Ebene. Es werden zunächst die Vektoren berechnet:

$$\begin{aligned} P_a &= P_{00} - P_{10} + P_{11} - P_{01} \\ P_b &= P_{10} - P_{00} \\ P_c &= P_{01} - P_{00} \\ P_d &= P_{00} \\ N_a &= P_a \times N \\ N_b &= P_b \times N \\ N_c &= P_c \times N \end{aligned}$$

Daraus entstehen die Hilfsvariablen:

$$\begin{aligned} u_0 &= (N_c \cdot P_d) \\ u_1 &= (N_a \cdot P_d) + (N_c \cdot P_b) \\ u_2 &= (N_a \cdot P_b) \\ v_0 &= (N_b \cdot P_d) \\ v_1 &= (N_a \cdot P_d) + (N_b \cdot P_c) \\ v_2 &= (N_a \cdot P_c) \end{aligned}$$

Diese skalaren Werte werden nun in die nachfolgenden Gleichungen eingesetzt, wenn $u_2 \neq 0$ ist:

$$\begin{aligned} Q_{ux} &= \frac{N_a}{2u_2} \\ u_x &= -\frac{u_1}{2u_2} \\ Q_{uy} &= -\frac{N_c}{u_2} \\ u_y &= \frac{u_0}{u_2} \end{aligned}$$

Die so erhaltenen Vektoren Q_{ux} und Q_{uy} sowie die skalaren Werte u_x und u_y werden dann in die nachfolgende Formel eingesetzt, wobei P der errechnete Schnittpunkt mit der Ebene ist:

$$\begin{aligned} a &= u_x + (Q_{ux} \cdot P) \\ b &= u_y + (Q_{uy} \cdot P) \\ u_0 &= a - \sqrt{a^2 - b} \\ u_1 &= a + \sqrt{a^2 - b} \end{aligned}$$

Falls $a^2 - b > 0$ ist, liegt der Schnittpunkt außerhalb des Vierecks, wenn nicht, berechnet man zuerst u_0. Liegt dieser im Bereich $0 \ldots 1$, so kann man ihn direkt verwenden, ansonsten wird u_1 verwendet, wenn dieser zwischen 0 und 1 liegt. Ist das nicht der Fall, so befindet sich der Schnittpunkt nicht innerhalb des Vierecks.

Für $u_2 = 0$ verlaufen die u-Achsen parallel und der Wert u wird direkt berechnet:

$$u = \frac{(N_c \cdot P) - u_0}{u_1 - (N_a \cdot P)}$$

Zunächst muß auch hier der Nenner auf ungleich Null geprüft werden, danach der Bereich für u.

Entsprechend muß man für die Berechnung von v vorgehen. Für $v_2 \neq 0$ gilt dann:

$$\begin{aligned} Q_{vx} &= \frac{N_a}{2v_2} \\ v_x &= -\frac{v_1}{2v_2} \\ Q_{vy} &= -\frac{N_b}{v_2} \\ v_y &= \frac{v_0}{v_2} \end{aligned}$$

Damit lassen sich die Koeffizienten a und b berechnen und einsetzen:

$$\begin{aligned} a &= v_x + (Q_{vx} \cdot P) \\ b &= v_y + (Q_{vy} \cdot P) \\ v_0 &= a - \sqrt{a^2 - b} \\ v_1 &= a + \sqrt{a^2 - b} \end{aligned}$$

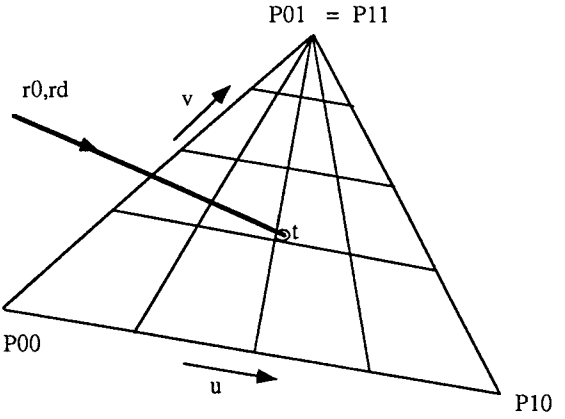


Abb. 19.25 Wie man aus einem Viereck ein Dreieck erzeugen kann

Für $v_2 = 0$, wenn also die v-Achsen parallel liegen, wird die nachfolgende Formel verwendet:

$$v = \frac{(N_b \cdot P) - v_0}{v_1 - (N_a \cdot P)}$$

Benötigt man die Texturkoordinaten u und v nicht in dieser Form, so ist es weniger rechenaufwendig, wenn man das Viereck zunächst in zwei Dreiecke zerlegt und dann mit der Formel aus dem nächsten Abschnitt arbeitet.

19.2.3.4 Schnitt: Dreieck und Strahl

Mit dem Verfahren für Vierecke lassen sich auch Dreiecke bearbeiten. Abb. 19.25 zeigt eine Variante. Die beiden Punkte P_{01} und P_{11} werden dazu einfach aufeinander gelegt. Im Punkt P_{01} (bzw. P_{11}) wird dann der Nenner der folgenden Gleichung zu Null.

$$u = \frac{(N_c \cdot P) - u_0}{u_1 - (N_a \cdot P)}$$

Man wählt dann z.B. den Wert 0 als Ergebnis für u. Liegen u und v zwischen 0 und 1, so trifft der Strahl das Innere des Dreiecks und es wird sichtbar. Jedoch lassen sich Dreiecke viel einfacher direkt behandeln.

Das Dreieck besteht aus den Punkten P_0, P_1 und P_2. Daraus läßt sich die Ebenengleichung direkt herleiten. Wir verwenden diesmal die explizite Form:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} P_{0x} \\ P_{0y} \\ P_{0z} \end{pmatrix} + a \cdot \begin{pmatrix} P_{1x} - P_{0x} \\ P_{1y} - P_{0y} \\ P_{1z} - P_{0z} \end{pmatrix} + b \cdot \begin{pmatrix} P_{2x} - P_{0x} \\ P_{2y} - P_{0y} \\ P_{2z} - P_{0z} \end{pmatrix}.$$

Wenn $a + b < 1$ und $a \geq 0$ sowie $b \geq 0$ gilt, dann liegt der Punkt auf der Ebene innerhalb des vorgegebenen Dreiecks. Diese Gleichung muß nur noch mit der Strahlgleichung

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} + t \cdot \begin{pmatrix} x_d \\ y_d \\ z_d \end{pmatrix}$$

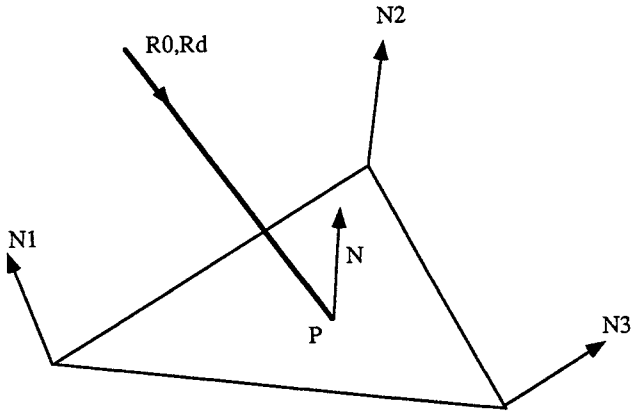


Abb. 19.26 Die Normalenvektoren für ein Dreieck mit scheinbar gekrümmten Flächen

gleichgesetzt werden. Es ergibt sich ein lineares Gleichungssystem mit drei Unbekannten, das, in Matrixform geschrieben, dann so aussieht:

$$\begin{pmatrix} P_{1x}-P_{0x} & P_{2x}-P_{0x} & -x_d \\ P_{1y}-P_{0y} & P_{2y}-P_{0y} & -y_d \\ P_{1z}-P_{0z} & P_{2z}-P_{0z} & -z_d \end{pmatrix} \cdot \begin{pmatrix} a \\ b \\ t \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix} - \begin{pmatrix} P_{0x} \\ P_{0y} \\ P_{0z} \end{pmatrix}$$

Daraus lassen sich a, b und t bestimmen.

In unserem Raytracer ist die Schnittberechnung für Dreiecke ähnlich diesem Verfahren implementiert. Dabei wird eine Hilfsmatrix vorausberechnet, die dann bei der Schnittberechnung verwendet werden kann. Sie enthält zusätzlich eine Vektorkomponente, die senkrecht zur Ebene ist. In `inter_triangle` wird die inverse Matrix bestimmt und die Schnittpunktberechnung durchgeführt. Die inverse Matrix der obigen Gleichung läßt sich nicht im voraus berechnen, denn R_d ist erst bei der Schnittbestimmung bekannt. Der Algorithmus stammt aus einem Public-Domain-Raytracer und hat sich als recht brauchbar erwiesen.

Will man Dreiecke einsetzen, um größere Objekte zu erzeugen, so ist es günstiger, Normalenvektoren an jedem Eckpunkt zu ermöglichen. Dann kann der Shader zwischen diesen Werten interpolieren. Die Normalen werden für die Schnittpunktberechnung nicht verwendet, sondern nur, wenn der Shader die Normale am Schnittpunkt anfordert. Dazu wird in der Routine `inter_triangle` eine Interpolation durchgeführt. Abb. 19.26 zeigt ein Schema dazu. Die Dreiecksfläche wirkt dann in der Darstellung gebogen. Da bei spiegelnden Flächen auch die Normale verwendet wird, lassen sich dadurch auch spiegelnde gekrümmte Oberflächen erzeugen. Somit steht der Konstruktion von Zerrspiegeln nichts mehr im Wege.

19.2.3.5 Schnitt: Polygon und Strahl

Es gibt dazu zahlreiche Verfahren, zum Beispiel die Zerlegung. Ein Polygon wird dazu einfach in Dreiecke zerlegt. Abb. 19.27 zeigt das Verfahren. Die Drei-

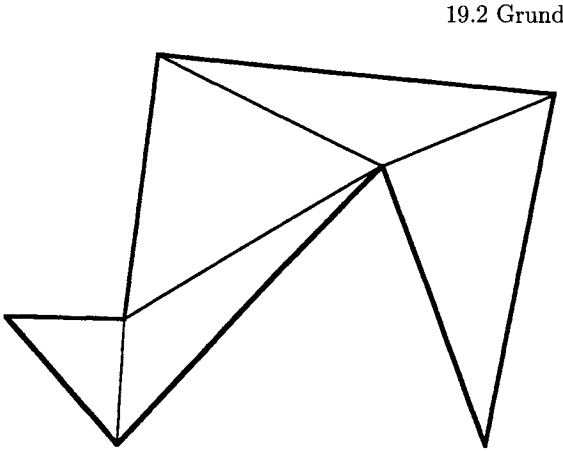


Abb. 19.27 Die Zerlegung eines Polygons in Dreiecke

ecke können danach dem Raytracer zugeführt werden. Dadurch ist man auch nicht mehr auf planare Polygone angewiesen, sondern kann beliebige im Raum liegende Formen verwenden.

Ein direkt mit Polygonen arbeitender Algorithmus prüft zunächst den Schnitt zwischen Sichtstrahl und der Ebene, in der das Polygon liegt (nur wenn dieses planar ist). Nach einer Koordinatentransformation muß nur noch geprüft werden, ob die lokalen Koordinaten des Schnittpunkts im Polygon sind. Verfahren dazu sind von den 2D-Algorithmen bekannt.

19.2.4 Rotationskörper

Für alle Körper mit Rotationssymmetrie läßt sich sowohl das Konstruktionsverfahren als auch die Darstellung beim Raytracer vereinfachen. Ein Rotationskörper wird zum Beispiel durch Eingabe der Randlinie definiert. Abb. 19.28 zeigt ein Beispiel. Hier wird der Umriß einer Vase definiert, die in Abb. 19.29 plastisch dargestellt ist.

Nun könnte man diesen Rotationskörper in ein polygonales Objekt umrechnen, doch dann erhält man sehr viele Polygone bzw. Drei- oder Vierecke. Außerdem bleibt das Problem der Kanten, da man nicht eine genaue Kreisfläche verwenden kann, sondern diese ebenfalls mit einem Polygon annähern müßte. Für das Raytrace-Verfahren gibt es jedoch einen eleganteren Weg. Da die Randkurve meist aus Geradenstücken zusammengesetzt ist, genügt es, diese zur Definition von Kegelstümpfen zu verwenden. Abb. 19.30 zeigt die zerlegte Vase. Bei der Zerlegung werden die Geradenstücke als Rand für die Kegelstümpfe verwendet. Die Kegelstümpfe werden dann unabhängig voneinander dem Raytracer als Objekte zugeführt.

Die Übergänge zwischen den Kegelstümpfen sind nach dem Raytracen noch sichtbar, da der Normalenvektor an diesen Stellen springt. Eine Verbesserung wäre es, bei den Kegelstümpfen auch interpolierte Normalenvektoren zuzulassen, die als zusätzliche Attribute mit angegeben werden. Für die Shader-

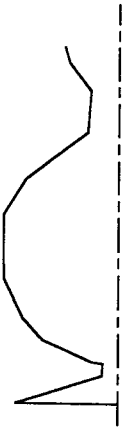


Abb. 19.28 Die Randkurve für einen Rotationskörper

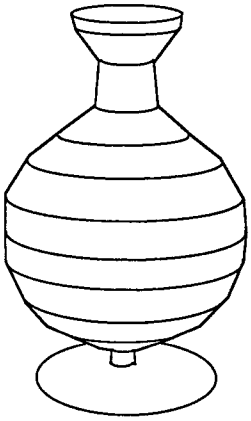


Abb. 19.29 Die Vase entsteht nach dem Rotieren der Randkurve um eine Symmetrieachse

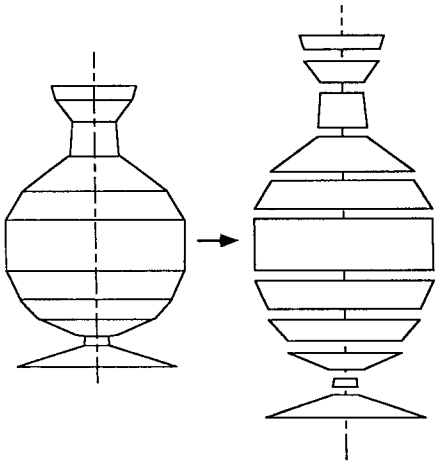


Abb. 19.30 Zerlegen eines Rotationskörpers in Kegelstümpfe

Abb. 19.31 Die fertige Vase nach dem Schattieren

Berechnung könnte man diese Normalen beim Auftreffen auf die Mantelflächen verwenden, ähnlich zu der Normaleninterpolation bei Dreiecken, so wie es in dem Raytracer auf Diskette implementiert ist. Abb. 19.31 zeigt die fertige Vase dreidimensional.

Liegt eine Definition als mathematische Kurve vor, so könnte man auch eine direkte Schnittberechnung durchführen. Dafür gibt es eine interessante Möglichkeit. Der Verlauf des Sichtstrahls wird in ein lokales Koordinatensystem umgerechnet. Dann kann die Schnittberechnung für 2D-Systeme durchgeführt werden. Dadurch ergibt sich eine Gleichung zweiten Grades für den Sichtstrahl. Nun muß die Gleichung für die Randkurve eingesetzt werden. Wenn die Kurve ebenfalls durch eine quadratische Gleichung dargestellt wird, so muß anschließend eine Gleichung vierten Grades gelöst werden. Bei Kurven höherer Ordnung, die für einen komplizierteren Kurvenverlauf notwendig werden, sind dann nur noch Approximationsverfahren zur Lösung möglich. Eine Zerlegung in Geradenstücke und die Übertragung in Kegelstümpfe ist in jedem Fall vorteilhafter.

19.3 Konstruktive Geometrie und Raytracing: CSG

Wenn man komplizierte geometrische Objekte konstruieren will, so verwendet man einfache Grundkörper, die man zu komplexeren Objekten zusammensetzt. Dabei wird eine Vereinigung der Objekte durchgeführt. Dies entspricht einem logischen ODER. Alle Raytracer beherrschen das „ODER" von Objekten. Interessant ist es aber auch, Schnittmengen (also „UND") oder gar Differenzen zu verwenden. Dann ist es möglich, „Löcher" in ein Objekt zu stanzen, indem man z.B. einen Zylinder von einer Kugel subtrahiert. Übrig bleibt dann ein zylindrisches Loch in der Kugel.

Diese Möglichkeiten werden unter dem Begriff CSG zusammengefaßt. S.D. Roth beschrieb 1982 ein Verfahren, mit dem Raytracer relativ einfach CSG implementieren können. Abb. 19.32 zeigt ein komplexes Objekt, das mit CSG

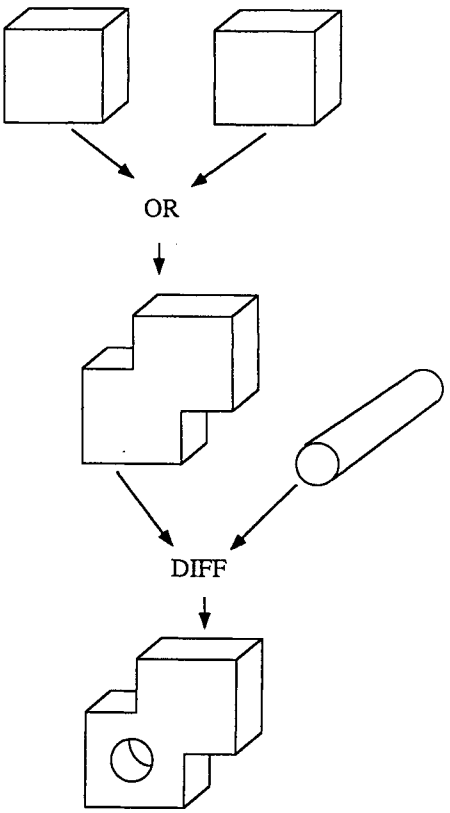


Abb. 19.32 Die CSG-Operationen ODER und DIFF

konstruiert wurde. Dabei werden zwei Würfel durch ODER verknüpft, also die Vereinigungsmenge gebildet. Dann wird von diesem kombinierten Objekt ein Zylinder subtrahiert. Unten sieht man das entstandene Gebilde. Abb. 19.33 zeigt den Vorgang bei der UND-Bildung. Zwei Scheiben (Zylinder) werden hier mit UND verknüpft. Durch die Bildung von Schnittmengen lassen sich sehr interessante neue Objektformen kreieren. So ist es möglich, durch Verknüpfung von einer Kugel mit einem Würfel abgerundete Kanten beim Würfel zu erzeugen. Die Subtraktion wird hingegen verwendet, um Löcher in Objekte zu stanzen. Dabei können diese Objekte selbst wieder aus komplexen Grundobjekten zusammengesetzt werden. Wichtig ist dabei, daß das resultierende Objekt komplett mit Raytracing behandelt werden kann.

Zunächst einmal muß man sich alle Schnittpunkte eines Objekts mit dem Sichtstrahl merken. Für die bisher bekannten Grundkörper wie Kugel, Zylinder oder allgemein Quadriken ist das kein Problem, es gibt dabei maximal zwei Schnittpunkte. Auch Würfel und konvexe Objekte können damit direkt behandelt werden. Dies ist auch der Grund, warum die bisher behandelten Raytracing-Prozeduren immer zwei Parameter mitgeführt haben: `t1` und `t2` in der Datenstruktur `hitdata`. Diese beiden Schnittpunkte, sowie die Informa-

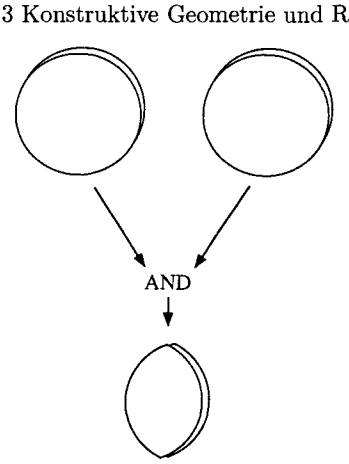


Abb. 19.33 Die CSG-Operation UND

tion, ob der Sichtstrahl eingedrungen ist oder herauskam, werden nun für die Verknüpfung ausgewertet. Abb. 19.34 zeigt das sogenannte Roth-Diagramm, das zur Berechnung der Verknüpfungen verwendet werden kann.

Hier sind, zur Vereinfachung zweidimensional, zwei Grundkörper dargestellt. Bei A handelt es sich um einen Quader und B soll ein Zylinder sein. Unter den Objekten ist der Verlauf der Sichtgeraden eingezeichnet. Dabei sind die Schnitte mit den einzelnen Objekten markiert und das Innere als durchgezogene Linie dargestellt. Der Sichtstrahl tritt bei A am Schnittpunkt t_1 in das Objekt ein und bei t_2 wieder heraus. Soll nun Objekt B subtrahiert werden, so muß das neue Paar der Schnitte berechnet werden. Dabei wird der erste Schnittpunkt von t_1 auf t_4 nach hinten verlegt, denn erst dort tritt der Strahl in das Objekt ein. Der Schnitt t_2 bleibt als Austrittsort erhalten. Hier sehen Sie auch, wozu man den zweiten Schnittpunkt benötigt. Bei Körpern, die subtrahiert werden, kehren sich praktisch Ein- und Austritt in ihrer Wirkung um. Bei der Vereinigung wird der erste Schnitt von B und der letzte Schnitt von A weitergereicht. Bei der Schnittmenge wird der erste Schnitt von A zum Eintrittsort und bei t_4 ist Objektende. Auch hier benötigt man wieder den zweiten Schnitt als Information zur Mengenbildung.

So gibt es für jede mögliche Kombination von Eintritt und Austritt des Sichtstrahls und für jede Operation eine Vorschrift. In diesem Fall war der Beobachter außerhalb der Körpers. Man muß nochmals alle Fälle betrachten, wenn der Ausgangsort des Sichtstrahls innerhalb eines Körpers ist, was z.B. bei durchsichtigen Körpern der Fall sein kann, oder bei einem Schattenstrahl, wenn z.B. ein Strahl im Fall A DIFF B von t_4 aus in Richtung Lichtquelle geschickt wird. Der Strahl durchquert dann den Körper B, der hier als Subtraktionskörper dient.

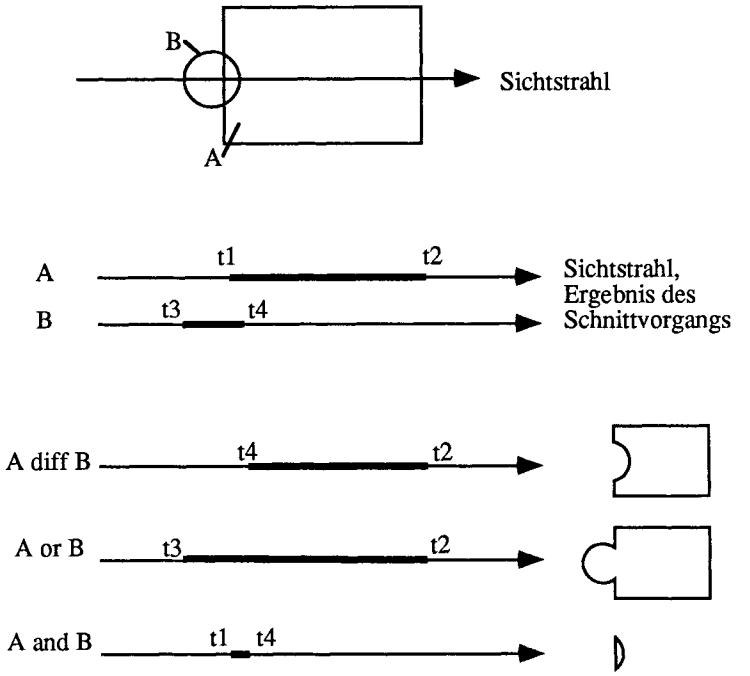


Abb. 19.34 Das Roth-Diagramm für CSG-Operationen

19.3.1 Der CSG-Algorithmus

Die CSG-Operationen UND, ODER und DIFF sind in dem Beispiel-Raytracer, der sich auf der Diskette befindet, implementiert. Daraus wollen wir an dieser Stelle einen kleinen Teil besprechen.

Die Funktion `intersect` prüft die Schnittpunkte für alle Objekte der Szene. Als Parameter wird dazu ein Pointer auf die Objekte `primpp` und der Sichtstrahl übergeben. Als Ergebnis liefert die Routine `TRUE`, wenn ein Objekt gefunden wurde und in `hitdata` steht dann, welches Objekt getroffen wurde.

```
FUNCTION intersect(primptr : primpp; ray : rayrec;
         VAR hitdata : isectrec) : BOOLEAN;
VAR
  hitleft,hitright : isectrec;
  flag,flagl,flagr : BOOLEAN;
  mindist          : DOUBLE;
BEGIN
```

Zunächst wird angenommen, daß kein Objekt getroffen wird. `intersect` wird rekursiv aufgerufen, um den CSG-Baum abzuarbeiten.

```
  intersect := FALSE; (* not hit *)
  IF primptr <> NIL THEN BEGIN
```

Es gibt dabei zwei Möglichkeiten. Entweder es handelt sich um ein Grundobjekt, dann kann direkt geschnitten werden, oder es werden die CSG-Routinen aufgerufen.

```
    IF primptr^.csgop = csgprim THEN BEGIN
```

Für jedes Grundobjekt wird an dieser Stelle die objektspezifische Schnittroutine aufgerufen.

```
      CASE primptr^.objectart OF
        sphere:   BEGIN
                    flag := inter_sphere(primptr,ray,hitdata);
                    IF flag THEN intersect := TRUE;
                  END;
        cone :    BEGIN
                    flag := inter_cone(primptr,ray,hitdata);
                    IF flag THEN intersect := TRUE;
                  END;
       triangle : BEGIN
                    flag := inter_triangle(primptr,ray,hitdata);
                    IF flag THEN intersect := TRUE;
                  END;

        ... usw...

      END; (* case *)
    END ELSE BEGIN (* csg found *)
```

Hier sind die speziellen CSG-Routinen eingebaut. Jedes Verbundobjekt hat einen linken und rechten Zweig. Die Unterbäume repräsentieren wieder Objekte, die entweder weitere CSG-Objekte oder Grundobjekte darstellen. Der linke Pfad wird immer aufgerufen. Beim rechten Pfad ist es aber nicht mehr unbedingt nötig – wie sich noch zeigen wird.

```
(* erst mal nur linker Pfad *)
flagl := intersect(primptr^.left,ray,hitleft,maxdist);
```

`flagl` ist `TRUE`, wenn der linke Pfad ein Objekt im Sichtstrahl findet.

```
CASE primptr^.csgop OF
  csgor: (* or two objects *)
```

An dieser Stelle wird die ODER-Verknüpfung aufgerufen, bei der es notwendig ist, auch den rechten Pfad zu durchlaufen, denn es muß danach untersucht werden, welches das nähere Objekt ist.

```
BEGIN
  flagr := intersect(primptr^.right,ray,hitright,maxdist);
  IF flagl AND flagr THEN BEGIN
```

Beide Pfade hatten ein Objekt im Sichtstrahl.

```
IF hitleft.t < hitright.t THEN BEGIN
```

Es folgt der Code für den Fall, daß der linke Treffer näher war. Sonderfälle des CSG müssen unterschieden und einzeln ausgewertet werden.

```
hitdata.t := hitleft.t;
hitdata.enterflag := hitleft.enterflag;
hitdata.primptr := hitleft.primptr;
hitdata.usenormal := hitleft.usenormal;
hitdata.normalvec := hitleft.normalvec;
IF (hitleft.t2 < hitright.t2) THEN BEGIN
```

Auch der zweite Schnittpunkt muß betrachtet werden:

```
  hitdata.t2 := hitright.t2; (* aussenpunkt nehmen *)
  hitdata.enter2flag := hitright.enter2flag;
  hitdata.prim2ptr := hitright.prim2ptr;
  hitdata.use2normal := hitright.use2normal;
  hitdata.normal2vec := hitright.normal2vec;
END ELSE BEGIN (* hitleft.t2 >= hitright.t2 *)
```

Und nun entsprechend für alle anderen Fälle (siehe Diskette).

```
... ende csgor ...

END;
csgand: (* and two objects *)
```

Für die UND-Verknüpfung muß der rechte Zweig nur dann berechnet werden, wenn der linke einen Treffer hat, was unter Umständen eine ganze Menge Rechenzeit spart. Ferner kann man die UND-Verknüpfung auch verwenden, um hierarchische Objektstrukturen aufzubauen. Eine Kugel, die um ein komplexes Objekt herum liegt, kann man so in den linken Pfad setzen. Das Objekt im rechten Pfad wird nur dann untersucht, wenn der Sichtstrahl die Kugel trifft. Wenn man dieses Verfahren konsequent anwendet, so kann man eine Unmenge an Zeit sparen. Es ist allerdings nicht ganz trivial, die umhüllenden Kugeln automatisch zu berechnen.

```
BEGIN
IF flagl THEN BEGIN (* nur dann rechten Ast schneiden *)
  flagr := intersect(primptr^.right,ray,hitright,maxdist);
  IF flagr THEN BEGIN
    ...
```

Nur wenn sich beide Objekte schneiden, kann man die Schnittmenge ermitteln (siehe Diskette).

```
END; (* csgand *)
csgdiff: (* diff two objects *)
```

Bei der Differenzbildung kann man sich die Berechnung des rechten Pfades auch sparen, wenn der linke keinen Schnitt liefert, denn von einem leeren Objekt kann man nichts mehr subtrahieren, die Menge bleibt leer.

```
BEGIN
  IF flagl THEN BEGIN (* nur dann rechten Ast subtr. *)
    flagr := intersect(primptr^.right, ray,
                hitright, maxdist);
    IF flagr THEN BEGIN
```

Erst hier ist es nötig, Fallunterscheidungen zu treffen.

```
    ... usw ...

    END; (* csgdiff *)
  END; (* case *)
END;
END; (* end if *)
END;
```

Damit endet der Schnittalgorithmus. Er kann sicher noch weiter verbessert und durch weitere Verknüpfungen ergänzt werden. Auch ist noch offen, welche Farbe die subtrahierte Fläche bekommt, usw. Für Modifikationen des Beispielprogramms bleiben viele Möglichkeiten.

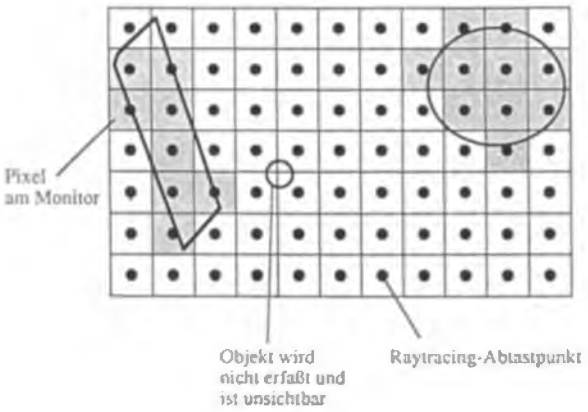


Abb. 19.35 Abtasten von Objekten mit grobem Gitter führt zum Verlust von Informationen

19.4 Verbesserte Raytracing-Techniken

Die Sichtstrahlen durch den Bildschirm werden bei unserem Raytracer durch ein Gitter hindurchgeschickt. Die Strahlen selbst sind (mathematisch betrachtet) unendlich dünn. Dadurch ergeben sich einige unerwünschte Effekte. Abb. 19.35 zeigt ein Schema dazu. Durch das Gitter entstehen unschöne Rastereffekte, wie man sie von 2D-Algorithmen (z.B. Linienzeichnen nach Bresenham) her kennt. Man nennt sie Aliasing-Effekte. Ein Strahl verfehlt ein Objekt nur knapp und liefert dann die Farbe des Hintergrunds. Die durch das Raster entstehenden scharfen Treppen wirken auf den Betrachter sehr störend. Die Objektform wird außerdem nur unzulänglich wiedergegeben.

Eine höhere Bildschirmauflösung verringert den Effekt, der aber immer noch sichtbar bleibt. Das kleine Objekt in der Mitte wird überhaupt nicht erfaßt und bleibt für das Bild unsichtbar. Bei einer Animation ergibt sich ein weiterer Effekt: Wenn das kleine Objekt bewegt wird, kann es sein, daß dieses nicht immer durch einen Sichtstrahl erfaßt wird und so zu blinken scheint. Außerdem stimmt seine Helligkeit und Größe nicht. Wenn das Objekt erscheint, belegt es die Fläche eines ganzen Pixels, obwohl es vielleicht viel kleiner ist. Um diese unerwünschten Effekte zu vermeiden, gibt es Methoden, die man beim Raytracing anwenden kann. Leider kosten diese Methoden zusätzliche Rechenzeit.

19.4.1 Supersampling

Beim sogenannten Supersampling verwendet man zur Abtastung ein feineres Gitter. Die Helligkeit eines Punktes wird dann durch Mittelung der Abtastpunkte berechnet. So entstehen auch an Grenzlinien Mischfarben. Der Aliasing-Effekt wird dadurch reduziert. Die Übergänge von Objekten sind dadurch wesentlich weicher. Kleinere Objekte werden allerdings immer noch übersehen, aber nur, wenn sie kleiner als das neue Gitter sind. Damit ist auch dieser Ef-

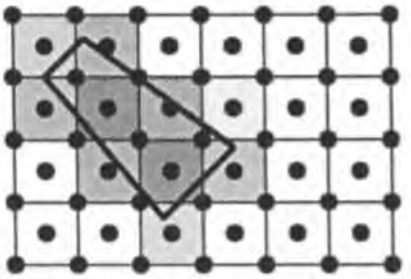


Abb. 19.36 Supersampling für Ränder ohne „scharfe" Kanten

fekt reduziert. Abb. 19.36 zeigt ein Gitter mit zusätzlichen Abtastpunkten. Die scharfen Kanten eines Objekts sind nicht mehr vorhanden. Der Übergang zur Umgebung ist weicher. Dieses Verfahren wird auch als „Antialiasing" bezeichnet.

Bei der Berechnung eines Bildpunktes verwendet man alle vier zusätzlichen Punkte, die um den früheren Abtastpunkt herumliegen. Dabei werden zum Beispiel die Helligkeitswerte aller Punkte pro Farbe aufaddiert und dann durch 5 geteilt. Dieser Wert bestimmt dann die Farbe auf dem Bildschirm. Ein Objekt, das kleiner als ein Pixel ist, erhält, wenn es von einem Abtaststrahl noch getroffen wird, nur noch ein Fünftel der Helligkeit. Für das Auge ergibt sich dadurch ein viel natürlicherer Eindruck.

Der Rechenaufwand erhöht sich bei diesem Verfahren linear mit der Anzahl der zusätzlichen Abtastpunkte. Um eine gute Wirkung zu erzielen, ist es nötig, sehr viele Abtastpunkte pro Pixel zu verwenden. Dadurch wird aber das sowieso schon langwierige Raytracing-Verfahren nicht mehr praktikabel.

19.4.2 Adaptives Supersampling

Hier werden nicht überall neue Sichtstrahlen hinzugefügt, sondern nur dort, wo sich ein größerer Helligkeitsunterschied ergibt. Dieses Verfahren bezeichnet man als „Adaptives Supersampling". Es wird zunächst das normale Abtastgitter verwendet. Erst wenn zwei benachbarte Pixel einen größeren Helligkeitsunterschied aufweisen, was zum Beispiel mit Hilfe eines Schwellenwerts festgestellt werden kann, werden weitere Sichtstrahlen ausgesendet. Dies kann man rekursiv fortführen und so an kritischen Stellen sehr viele Abtaststrahlen verwenden. Die Strahlen müssen dann ihrer Rekursionstiefe entsprechend gewichtet werden und die Helligkeit für das Bildschirmpixel wird daraus ermittelt. Dieses Verfahren führt zu sehr weich aussehenden Kanten.

Kleinere Objekte könnten hier aber wieder übersehen werden, wenn sie durch das Raster fallen. Das Grundraster sollte daher für Raytracing-Bilder,

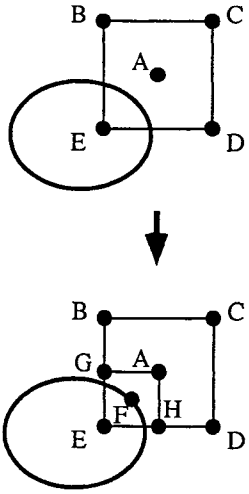


Abb. 19.37 Adaptives Supersampling vermeidet unnötige Rechenarbeit

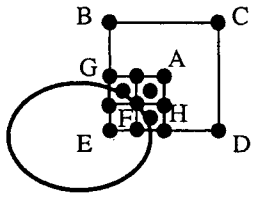


Abb. 19.38 Adaptives Supersampling mit Tiefe 3

die kleine oder weit entfernte Objekte enthalten können, auch das normale Supersampling einschließen. Dazu ein einfaches Beispiel in Abb. 19.37. Zunächst sind hier fünf Abtastpunkte A bis E vorhanden. Zwischen A und E besteht ein größerer Helligkeitsunterschied, weil hier eine Kugel für diesen Bildteil vorhanden sein soll. Dann werden neue Sichtstrahlen ausgeschickt und es entsteht die darunter abgebildete Situation. Die neuen Punkte F, G und H bieten nun genauere Information über das Objekt an dieser Stelle. Eine weitere Unterteilung ist möglich, denn zwischen A und F, G und F sowie F und H besteht immer noch ein Helligkeitsunterschied. Man müßte das Bild also nochmals unterteilen, wie Abb. 19.38 zeigt. Dabei wird davon ausgegangen, daß der Punkt F die Kugel noch getroffen hat. Die Unterteilung geschieht rekursiv und daher muß man die Tiefe auf einen vernünftigen Wert begrenzen. Für zwei Iterationsschritte sei hier die Formel für den zu errechnenden Helligkeitswert angegeben:

$$I = \frac{1}{4} \cdot \left(\frac{A+B}{2} + \frac{A+C}{2} + \frac{A+D}{2} + \frac{1}{4} \cdot \left(\frac{F+A}{2} + \frac{F+E}{2} + \frac{F+G}{2} + \frac{F+H}{2} \right) \right)$$

Entsprechend werden bei höheren Rekursionstiefen die einzelnen Sichtstrahlen gewichtet.

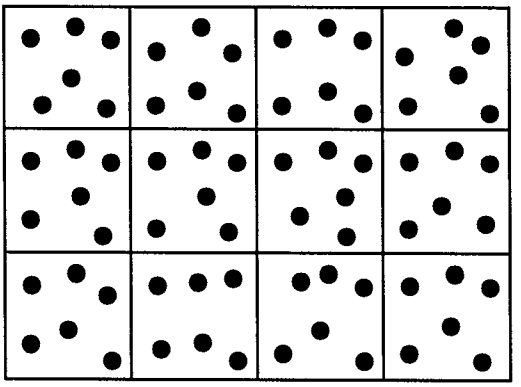


Abb. 19.39 Stochastisches Supersampling

19.4.3 Stochastic Sampling und Distributed Raytracing

Da bei den bisherigen Verfahren immer noch reguläre Gitter verwendet wurden, treten auch noch Aliasing-Effekte auf (wie zum Beispiel bei sehr feinen Gittern oder Objekten mit regulärer Struktur, die kleiner als das Abtastraster sind). Das Auge vermeidet solche Effekte dadurch, daß die Sehzellen nicht streng geometrisch angeordnet sind, sondern nur ungefähr in einem Gitter liegen. Die Verteilung der Sehzellen folgt statistischen Gesetzen und dies kann man sich auch beim Raytracing zur Verbesserung der Bildqualität zunutze machen. Dazu ordnet man ein Gitter beispielsweise so an, wie es Abb. 19.39 zeigt. Die Position der einzelnen Gitterpunkte kann man dann wie folgt berechnen:

```
X = xGitter + Rand() * Deltagitterx
Y = yGitter + Rand() * Deltagittery
```

Dabei sind `xGitter` und `yGitter` die Koordinaten des regulären Abtastrasters und `Deltagitterx` sowie `Deltagittery` der Abstand der Gitterpunkte. Die Funktion `Rand()` liefert einen Zufallswert zwischen 0 und 1. Dabei werden, wie bei den Supersampling-Verfahren, mehrere Abtaststrahlen pro Pixel verwendet und anschließend summiert.

19.4.3.1 Halbschatten

Mit einem stochastischen Verfahren ist es auch möglich, eine realistischere Darstellung von Schatten zu bekommen. Raytracer erzeugen normalerweise harte Schatten mit scharfen Grenzen. In der Realität sind die Grenzen von Schatten jedoch immer dann weich, wenn die Lichtquelle nicht punktförmig ist, sondern eine gewisse Ausdehnung besitzt. Da aber die Schattenbestimmung mit einem Strahl genau in das Zentrum der Lichtquelle erfolgt, wird der Schatten hart. Man kann das ändern, indem man einen Schattenstrahl in Richtung Lichtquelle sendet, der einen zufällig auf der Fläche der Lichtquelle ausgewählten Zielpunkt ansteuert. Zusammen mit Supersampling erhält man dann automatisch

Abb. 19.40 Halbschatten bei verbessertem Raytracing

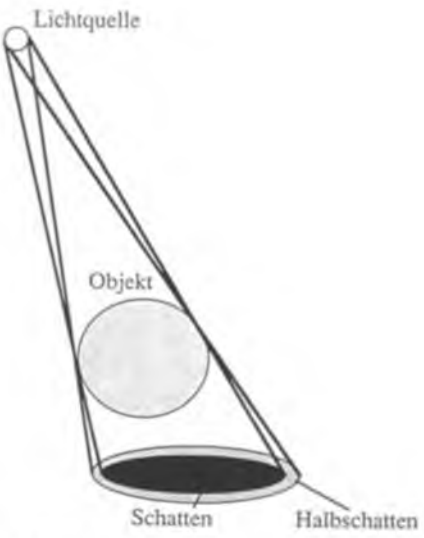


Abb. 19.41 Enstehung eines Halbschattens

Halbschatten mit der richtigen Ausdehnung. Abb. 19.40 zeigt eine Kugel mit Halbschatten auf dem Untergrund.

Abb. 19.41 zeigt ein Schema zur Entstehung von Halbschatten. Im Bereich des Halbschattens erreicht nur ein Teil des Lichts den Grund, denn ein Teil der Lichtquelle wird durch das Objekt abgedeckt. Daher entsteht ein Übergang vom Schattenbereich, in dem kein Lichtstrahl ankommt, bis zu dem Bereich, bei dem alle Lichtstrahlen ankommen. In unserem Beispiel-Raytracer auf der Diskette ist ein Einbau dieses Verfahrens vorgesehen, denn die Größe der Lichtquelle wird in der Datenstruktur mitgeführt.

Das Supersampling kann man in die Hauptschleife einbauen. Die Schattenstrahlroutine muß man dann entsprechend verändern, so daß der Strahl einen Punkt der Lichtquelle auswählt.

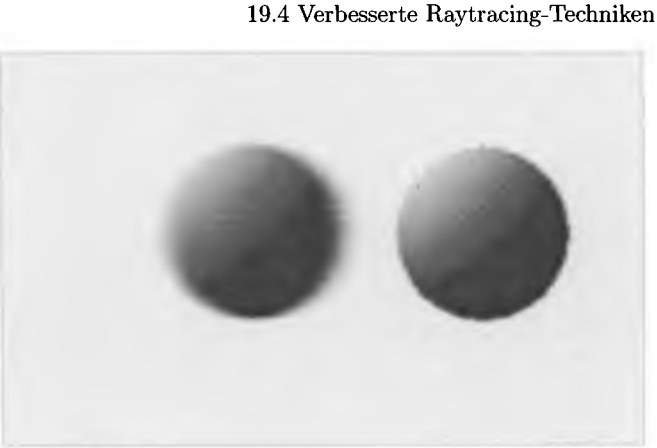


Abb. 19.42 Die linke Kugel zeigt eine Bewegungsunschärfe, das sogenannte „motion blur"

19.4.3.2 Bewegungsunschärfe (Motion Blur)

Bei der Animation mit bewegten Objekten im Raytracing-Verfahren treten ähnliche Probleme auf. Normalerweise würde für einen kurzen Zeitraum t die Bewegung eingefroren und dann das Bild berechnet. Dies führt in der Praxis zu sehr ruckartigen Bewegungen. Das Auge ist jedoch gewohnt, Bewegungsunschärfen zu sehen, da ein bewegter Gegenstand im Mittel weniger Licht in der Bewegungsrichtung auf den Augenhintergrund gelangen läßt. Abb. 19.42 zeigt ein Beispiel.

Die linke Kugel bewegt sich nach links und zeigt daher eine Bewegungsunschärfe. Die rechte Kugel verharrt in Ruhe und ist daher scharf abgebildet. Ein Raytracer kann diese Bewegungsunschärfen nachbilden, indem in kürzeren Zeitintervallen abgetastet wird, und man die Bilder wieder aufsummiert und mittelt. Auch hier sind wieder stochastische Methoden denkbar, um Aliasing-Effekte zu vermeiden, wie sie zum Beispiel auch bei bewegten Objekten auftreten.

Vielleicht kennen Sie alle diesen Effekt von Spielfilmen. In Western sieht man häufiger Speichenräder, die sich scheinbar rückwärts drehen, obwohl der Wagen vorwärts fährt. Das kommt daher, daß die Kamera immer nur zu einem kleinen Zeitpunkt das Bild aufnimmt, die Position der Speichen sich aber schneller ändert, als es die Kamera erfassen kann. Dadurch entsteht eine scheinbare Bewegung, die bei ungünstigem Zusammenfallen von Rotationsgeschwindigkeit und Abtastzeitpunkt der Filmkamera zu dem Rücklauf-Effekt führt. Beim Raytracing von bewegten Bildern können solche Effekte ebenfalls auftreten. Nicht nur bei Speichenrädern, sondern auch bei Gitterstrukturen und allgemein bei allen regulär angeordneten Objekten.

Um diese Effekte zu vermindern, muß der Abtastpunkt statistisch schwanken und es müssen mehr Zeitpunkte gewählt werden, als für die später ablaufende Bewegung nötig wären. Dies ist natürlich extrem rechenzeitaufwendig.

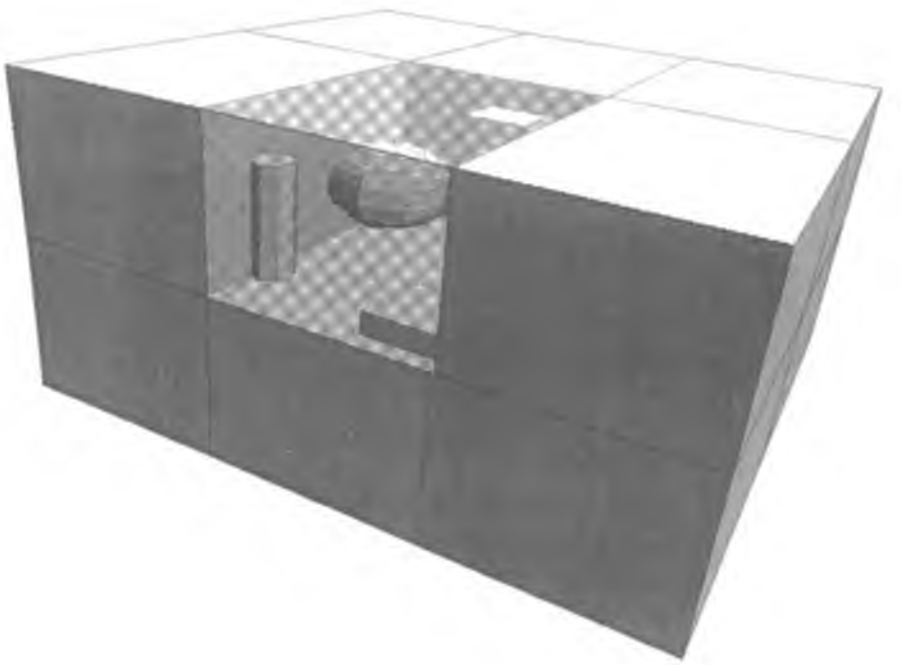


Abb. 19.43 Bounding Boxes zur Vermeidung von unnötigen Schnittpunktberechnungen

Daher wollen wir uns im nächsten Abschnitt mit Verfahren beschäftigen, welche die Rechenzeit vermindern helfen.

19.5 Beschleunigung von Raytracing-Verfahren

Zunächst kann man natürlich an den Algorithmen selbst Optimierungen durchführen, zum Teil durch geschickte Vorausberechnung von Parametern bereits beim Anlegen der Datenstrukturen, zum Teil aber durch mathematische Umformungen. Hier wollen wir aber noch radikalere Methoden zum Senken der Rechenzeit kennenlernen.

19.5.1 Bounding-Box-Algorithmen

Darunter versteht man Algorithmen, die den Raum, in dem sich die Objekte befinden, weiter unterteilen. Im „brute force" Raytracing wird der Sichtstrahl mit allen in der Szene befindlichen Objekten geschnitten. Dadurch wächst der Rechenaufwand linear mit der Anzahl der Objekte. Wenn man die Objekte gruppieren kann, so ist es möglich, diesen Aufwand zu reduzieren. Abb. 19.43 zeigt eine Möglichkeit dazu.

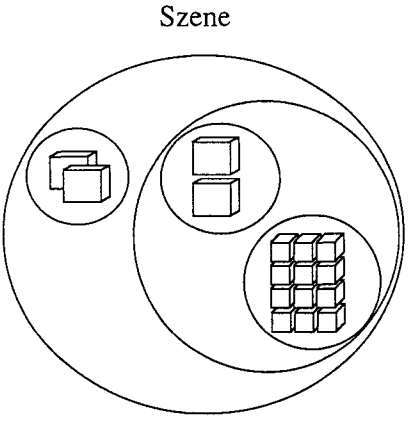


Abb. 19.44 Die Objekte werden durch Kugeln umschlossen

Der Raum wird dazu in Würfel oder Quader eingeteilt. Jedem Würfel ordnet man nun die Objekte zu, die innerhalb dieses Würfels liegen, oder von diesem geschnitten werden. Der Raytracer muß nun bei der Berechnung zunächst bestimmen, welche dieser Würfel vom Sichtstrahl geschnitten werden. Dies kann man z.B. mit einem modifizierten Bresenham-Algorithmus durchführen.

Dann werden alle vom Sichtstrahl getroffenen Würfel analysiert und die zuvor in entsprechenden Listen festgehaltenen Objekte, die zu diesen Würfeln gehören, mit dem Sichtstrahl nach dem schon beschriebenen Verfahren geschnitten. Alle Objekte in Würfeln, die vom Sichtstrahl nicht geschnitten wurden, kann man bei der Schnittberechnung ignorieren. Dadurch spart man sich enorme Rechenzeiten. Es ist allerdings nicht ganz einfach, diese Raumeinteilung konsequent durchzuhalten. So bleiben bei diesem einfachen Verfahren zahlreiche Fragen offen, z.B.: Wie groß sollen die Würfel gewählt werden? Was geschieht, wenn in einem Bereich wesentlich mehr Objekte vorhanden sind, als in anderen?

Der Algorithmus läßt sich noch erweitern, indem man unterschiedlich große Würfel verwendet, um damit den Raum hierarchisch zu unterteilen. Die Würfel werden dann umso feiner, je mehr Objekte sich dort befinden. Man erhält baumartige Datenstrukturen. Es ist dann allerdings nicht mehr so einfach, zu bestimmen, welche dieser Würfel noch vom Sichtstrahl getroffen werden, ohne hier wieder viel Rechenzeit zu investieren.

Statt mit Würfeln kann der Raum auch mit Kugeln unterteilt werden. Unser Raytracer ist dazu geeignet, hierarchische Strukturen mit Kugeln aufzubauen. Dazu verwendet man die CSG-Funktion UND. In die Szene werden zusätzliche Kugeln eingebaut, die immer eine Gruppe von nahe beieinanderliegenden Objekten umhüllen. In den linken Ast des CSG-Baums wird die umhüllende Kugel eingefügt. Der linke Ast deshalb, weil der Raytracer bei uns den linken Ast zuerst untersucht und den rechten Ast ignoriert, wenn der Sichtstrahl den linken Ast verfehlt. Abb. 19.44 zeigt eine Beispielszene. Alle Objekte in der Szene sind hier zur Unterscheidung als Quader gezeichnet. Diese werden von Kugeln umhüllt, die später bei der Darstellung unsichtbar bleiben. Diese

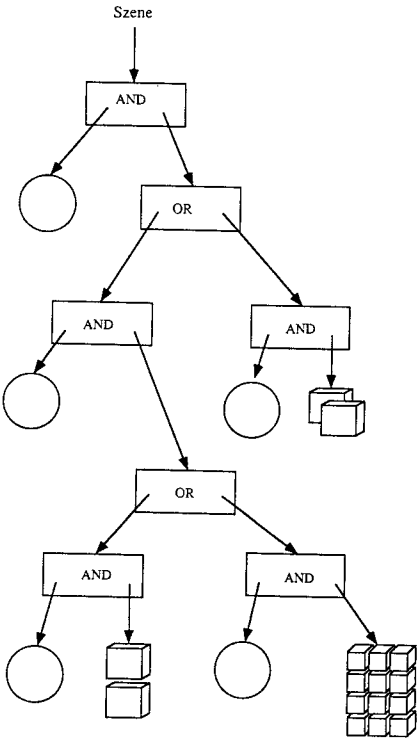


Abb. 19.45 Der dazugehörige CSG-Baum

Szene wird durch einen CSG-Baum in Abb. 19.45 beschrieben. Dabei werden die umhüllenden Kugeln immer im linken Ast gespeichert und die CSG-Operation UND angewandt. Gleichartige Teile sind durch die CSG-Operation ODER miteinander verknüpft.

Die erste, alles umschließende Kugel entscheidet zunächst, ob irgend ein vorhandenes Objekt geschnitten werden muß. Wird sie getroffen, so muß weiter untersucht werden. Es folgt dann ein ODER-Teil mit zwei Unterbäumen, wobei beide Bäume vom Raytracer untersucht werden müssen. Dort trifft man jeweils auf eine CSG-UND-Operation mit einem weiteren Unterbaum und einem Objekt. Der Raytracer prüft wieder zuerst den Schnitt mit den Testkugeln. Nur wenn dieser erfolgreich war, wird weiter untersucht.

Bei geschickter Aufteilung erreicht man, daß der Rechenaufwand nur noch logarithmisch mit der Zahl der Objekte ansteigt. Die zusätzlich zu testenden Kugeln seien dabei einmal vernachlässigt. In der Praxis wird man diesen Wert natürlich nicht erreichen, in jedem Fall ergibt sich jedoch eine starke Beschleunigung des Raytrace-Verfahrens.

19.5.2 Parallelisierung

Wenn man mehr als einen Prozessor für die Berechnung zur Verfügung hat, so ist es beim Raytrace-Verfahren besonders einfach, eine Parallelisierung zu erreichen. Jeder Prozessor bekommt dazu die gesamte Szenenbeschreibung. Ein Prozessor errechnet jedoch nur einen Teil des Gesamtbildes auf dem Bildschirm. Da sich jeder Bildpunkt unabhängig von den Nachbarn berechnen läßt, genügt es zur Parallelisierung, die äußerste Raytrace-Schleife auf die Prozessoren zu verteilen und dort die Parameter entsprechend zu belegen. Dabei muß man darauf achten, daß einzelne Prozessoren zu unterschiedlichen Zeiten fertig werden können. Diesen Prozessoren sollte man dann einen neuen Bildteil zuweisen, den sie dann berechnen können. Je nach Gesamtzahl der vorhandenen Prozessoren kann man das Bild zunächst einmal in einzelne Zeilen oder gar in Teile von Zeilen aufteilen. Bei einem Bild mit 640 x 480 Punkten und 100 Prozessoren wird man den Prozessoren einzelne Zeilen zuweisen. Ist ein Prozessor mit der Berechnung einer Zeile fertig geworden, so erhält er die nächste Zeile zugewiesen. Ein Hauptrechner sammelt die einzelnen Ergebnisse und zeigt sie am Bildschirm an.

Bei einer Animation ist es sogar möglich, die einzelnen Bewegungsphasen an unterschiedliche Prozessoren zu delegieren, da auch diese meist voneinander unabhängig sind, wenn man nicht zum Beispiel „Physical Modeling“ betreibt, also den Bewegungsablauf abhängig von physikalischen Interaktionen mit der Umgebung macht.

19.6 Public-Domain-Software

Für alle, denen der im Buch mitgelieferte Raytracer nicht ausreicht: Es gibt inzwischen eine Vielzahl öffentlich verfügbarer Software, die Sie für eigene Experimente nutzen können. Zwei der bekannteren sind der MTV-Raytracer und der DKBTrace.

Der *MTV-Raytracer*, geschrieben von Mark Terrence VandeWettering, ist ein Public-Domain-Raytracer mit C-Source-Code. Es handelt sich um einen kompakten Raytracer mit wichtigen Grundfunktionen, der sehr einfach zu portieren ist.

Persistence of Vision Raytracer, auch bekannt unter dem Namen DKBTrace bzw. jetzt unter dem Namen PV-Raytracer ist ebenfalls ein Public-Domain-Raytracer mit C-Source für unterschiedliche Plattformen. Das Programm ist extrem groß und ausgereift und enthält zahlreiche Raytracefunktionen (inklusive Torus etc.), einen komplexen Shader und Formatumwandlungsprogramme, z.B. zum Einlesen von Autocad-Dateien.

Farbtafeln

Abb. I Mit Hilfe von hierarchischer Modellierung kann relativ einfach ein kompliziert zusammengesetztes Objekt generiert werden, ohne daß die absoluten Koordinaten der Teilobjekte eingegeben werden müssen. Das Listing der zugehörigen Eingabedatei befindet sich im Kapitel „Grundlagen der dreidimensionalen Computergrafik".

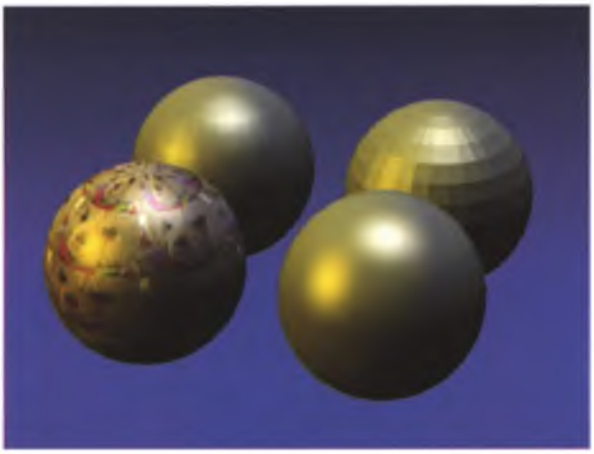
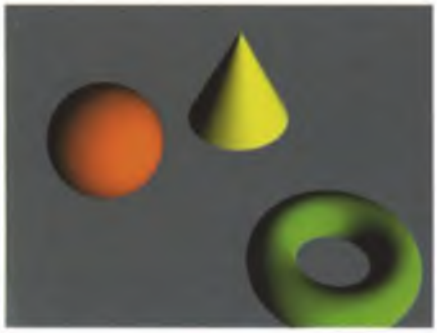


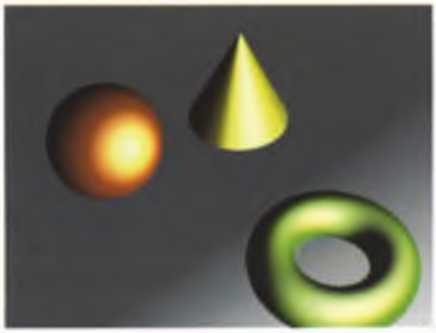
Abb. II Mit Hilfe verschiedener Schattierungsarten kann eine Kugel durch Polygone visuell mehr oder weniger gut angenähert werden. Die Verwendung einer Textur vergrößert die Detailvielfalt. Die hintere rechte Kugel ist mit Flat-Shading erzeugt, die hintere linke mit dem Gouraud-Algorithmus. Die vorderen Kugeln sind mit Phong-Shading dargestellt.

Abb. III Auswirkung verschiedener Lichtquellen sowie verschiedener Reflexionsmodelle: D1 bis D4 zeigen die Szene bei diffuser Reflexion, S1 bis S4 bei zusätzlicher spiegelnder Reflexion.
D1 bzw. S1 wurden mit einer ambienten, D2 und S2 mit einer Richtungs-, D3 und S3 mit einer Punktlichtquelle und D4 und S4 mit einem Strahler beleuchtet.
Bei ambienter Beleuchtung sind die Bilder identisch. Der weiße Punkt zeigt die Position von Punktlichtquelle bzw. Strahler an.

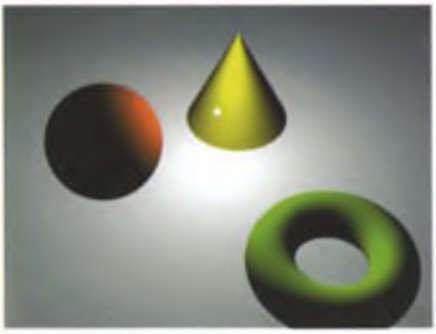
D1/S1



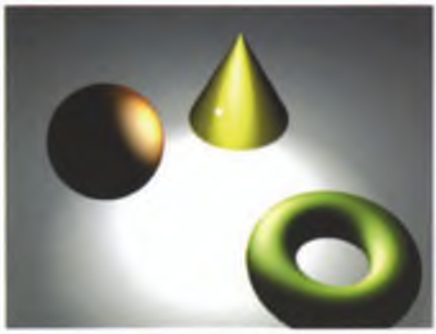
D2



S2



D3



S3

D4

S4

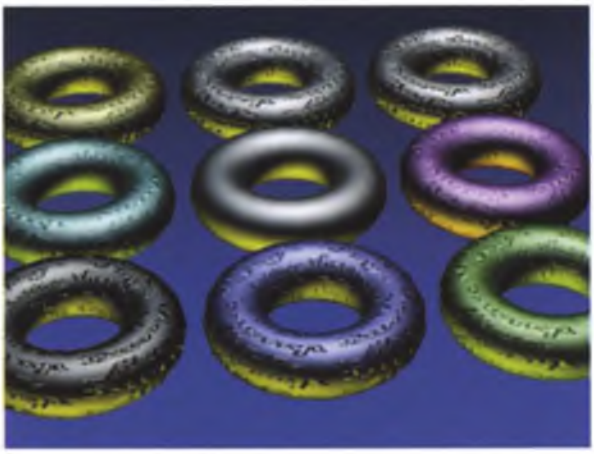


Abb. IV Moduliert man den Normalenvektor eines Objektes mit einer Textur, so wird das Objekt „dellig". Bei der abgebildeten Serie wurde die Stärke der Modulation variiert, so daß die Schrift mehr oder weniger plastisch wirkt.

Abb. VI Auch diese Szene wurde mit dem Radiosity-Verfahren berechnet. Ein detailliertes geometrisches Modell, der Einsatz von Texturen und die Erweiterung des Verfahrens um die Simulation von realen Lichtverteilungskurven führt zu extrem realistischen Bildern.
(© HL Technik, Mindware, UP↑Vision)

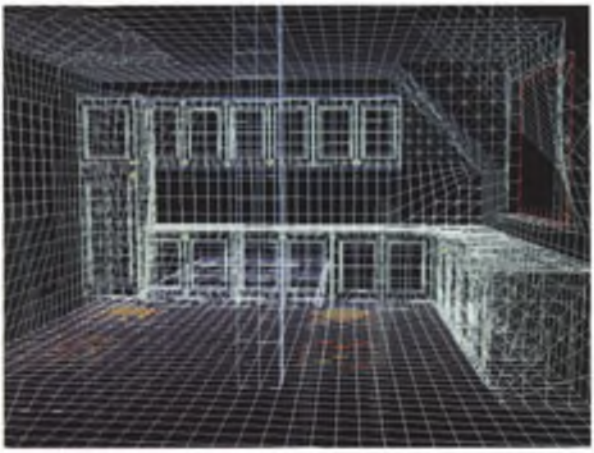


Abb. Va Das Drahtmodell der auf der folgenden Seite abgebildeten Küche. Durch die Aufteilung in viele kleine Flächenelemente ist die Energieberechnung auch bei ungleichmäßiger Lichtverteilung möglich.

Abb. Vb Foto einer „echten" Küche.

Abb. Vc, d Bilder einer Rendering-Küche mit lokalem Beleuchtungsmodell (oben) und einer Küche mit globaler Beleuchtungsberechnung. Die unterschiedlichen Lichtverhältnisse sind gut erkennbar. Die Rechenzeiten für das obere Bild betrug circa 2 Minuten und für das untere Bild circa 2 Stunden.

Abb. VIIa Wird zusätzlich zur Radiosity-Berechnung noch eine normale Beleuchtung berechnet, so sind auch Glanzstellen und Bumpmapping möglich.

Abb. VIIb Bei reiner Radiosity-Berechnung wirkt die gleiche Szene viel matter. Die Rechenzeit betrug für Bild a und Bild b circa eine Stunde. Beide bestehen aus 1409 Patches und 8180 Elementen.

Abb. VIII In dieser Bilderserie ist der Verlauf bei einer Radiosity-Iteration zu beobachten. Die Szene wird zunehmend heller, da mehr und mehr Energie auf die empfangenden Flächen verteilt wird.

VIIIa 4 Iterationen

VIIIb 24 Iterationen

VIIIc 2000 Iterationen

Abb. IX Durch die Einbeziehung eines ambienten Anteils ändert sich die mittlere Helligkeit während der Radiosity-Iteration nicht. Lediglich die Helligkeitsverteilung konvergiert gegen das Ergebnis.

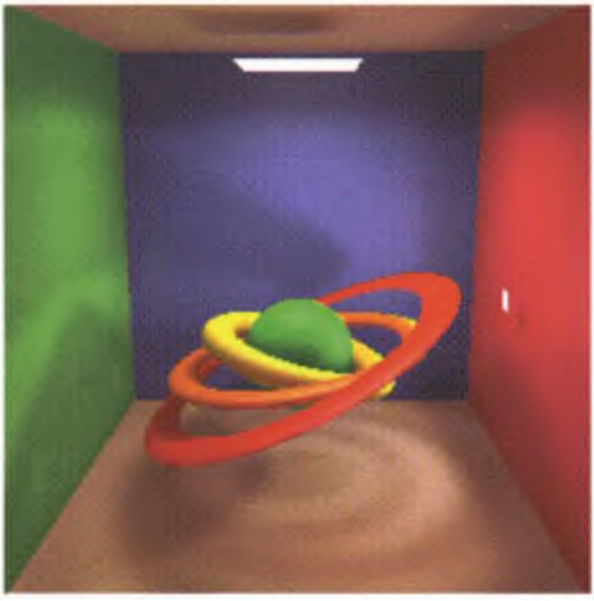
IXa 4 Iterationen

IXb 24 Iterationen

IXc 2000 Iterationen

Abb. X Diese Bilder zeigen den Unterschied bei der Verwendung einer Echtfarb-Grafikkarte (16,7 Millionen Farben) und einer VGA-Karte mit 256 Farben. Bei 256 Farben wird die Farbauflösung durch Dithering scheinbar erhöht.

Xa „Saturn" in Echtfarbendarstellung.



Xb „Saturn" gedithert in 256 Farben.

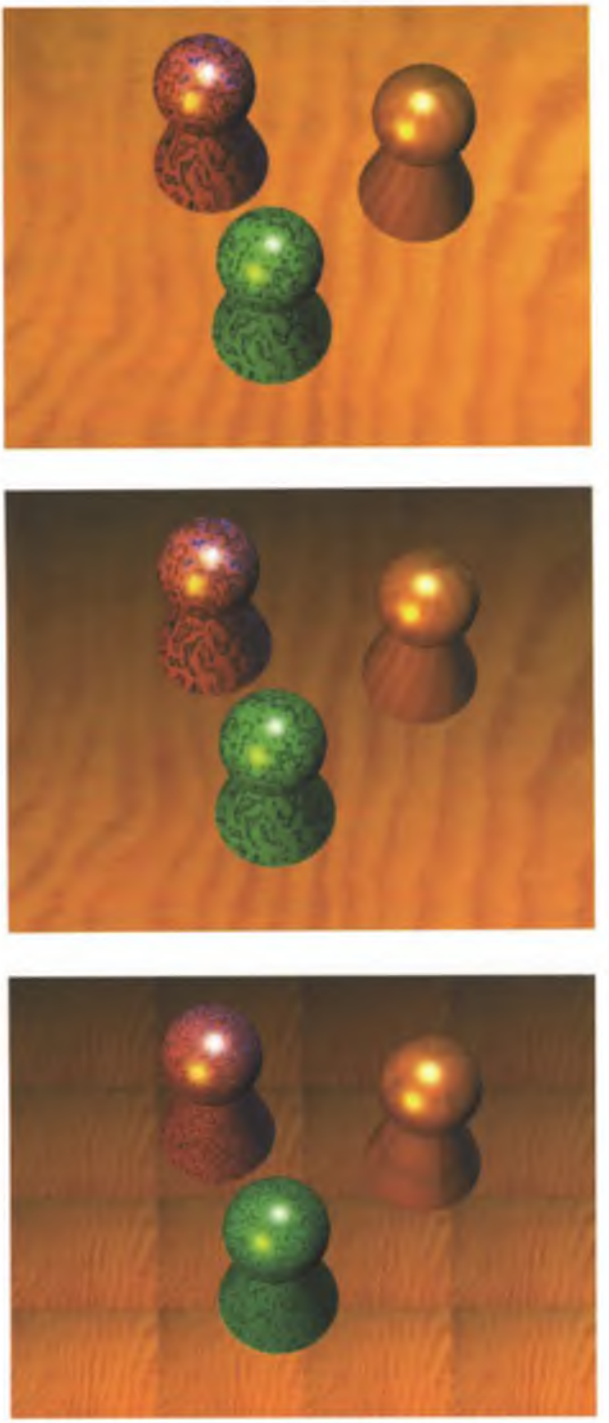


Abb. XIa–c Durch verschiedene Hintergründe können verschiedene Eindrücke erzielt werden.

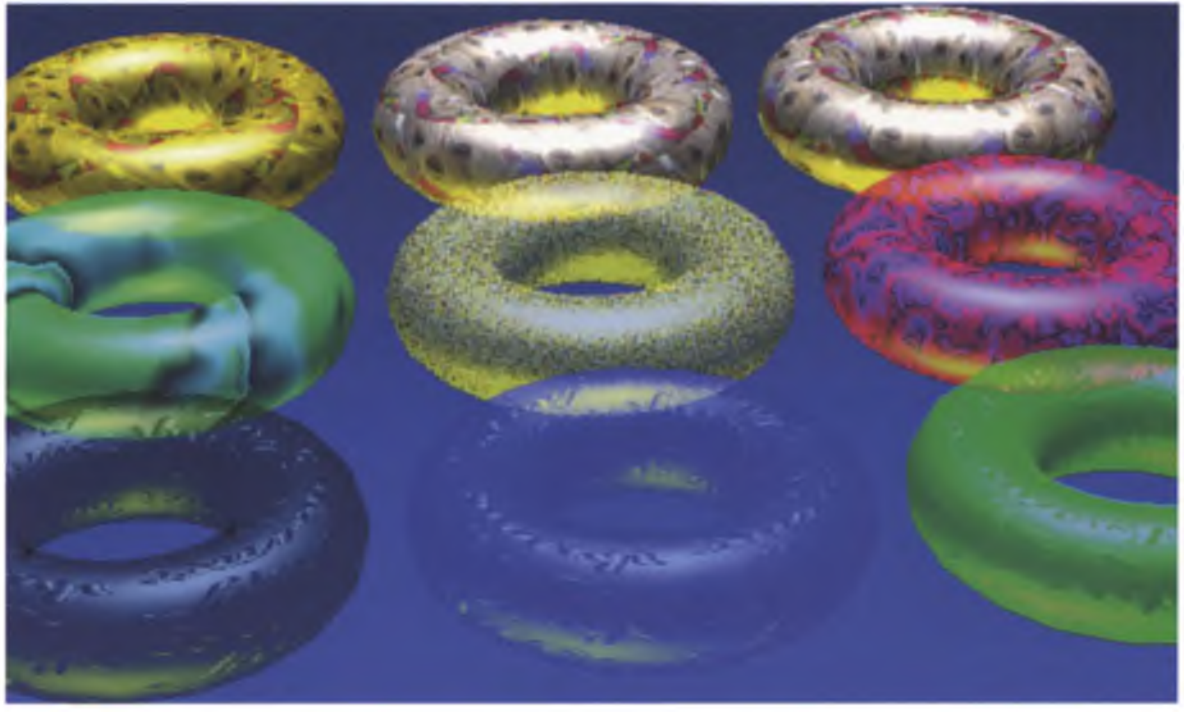


Abb. XII Verschiedene Texturen. Von hinten nach vorne sind zu sehen: 2D-Texturen, 3D-Texturen und Bumpmapping, jeweils kombiniert mit Transparenz. Das Bild wurde mit „normalem" Rendering erzeugt und benötigte weniger als zwei Minuten Rechenzeit.

Abb. XIII In dieser Szene ist das Licht „um die Ecke geflossen". Sie besteht aus einem L-förmigen Raum, von dem hier nur das lange Ende zu sehen ist. Die Lichtquelle befindet sich dagegen am Kopf des kurzen Endes.

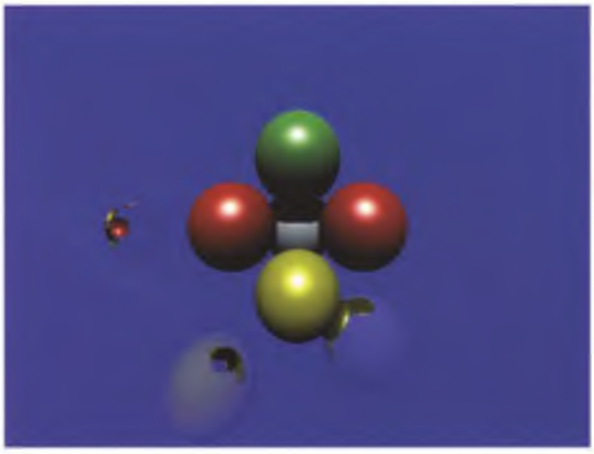


Abb. XIV Raytracing-Szene mit 8 Kugeln. Zwei der Kugeln sind verspiegelt, wobei der Grad der Verspiegelung unterschiedlich gewählt wurde. Die linke Kugel ist als durchsichtige Glaskugel ausgeführt, so daß sie wie eine Linse wirkt. Die Szene wird mit einer Lichtquelle beleuchtet.

Abb. XV Raytracing mit CSG. Für die zwei Objekte im Vordergrund wurden überlappende Kugeln mit unterschiedlicher Farbe durch UND verknüpft. Daher ist nur der gemeinsame Bereich sichtbar. Im Hintergrund befindet sich eine große Kugel, um den Schattenwurf sehen zu können. Die Szene wird von zwei Lichtquellen mit hellblauer und weißer Farbe beleuchtet.

Abb. XVI Raytracing mit CSG. Diesmal wurde die DIFFERENZ-Operation angewendet. Von einer Kugel wird die Kombination (ODER) dreier Kugeln subtrahiert, also K = A - (B ODER C ODER D).
D ist eine verkleinerte Version von A, die daher einen Hohlraum erzeugt.
B und C sind so groß wie A, jedoch im Raum versetzt.
Im Hintergrund ist eine weitere Kugel für die Beobachtung des Schattenwurfs zu sehen. Auch hier werden wieder zwei Lichtquellen zur Beleuchtung verwendet. Die weiße Lichtquelle steht rechts und die blaue leicht oben.

Abb. XVII Raytracing von Kegelstümpfen, die hier hohl dargestellt sind. Zur Beleuchtung werden wiederum zwei Lichtquellen verwendet.

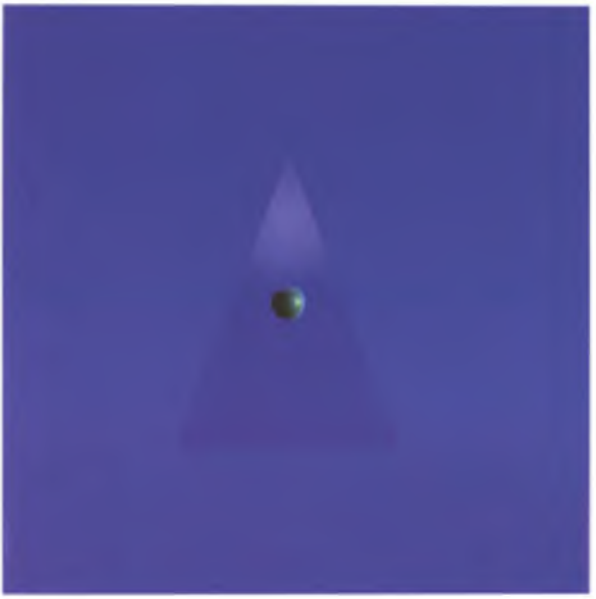


Abb. XVIII Raytracing eines Dreiecks. Es ist verspiegelt und nach „hinten gebogen" – die Normalen an den Eckpunkten zeigen von der Bildmitte weg. Vor dem Bildschirm befindet sich eine Kugel, die sich im Dreieck spiegelt. Da der Beobachter für den Raytracer unsichtbar ist, sind solche Spiegelungen möglich.
Zur Beleuchtung werden eine blaue und eine weiße Lichtquelle verwendet.

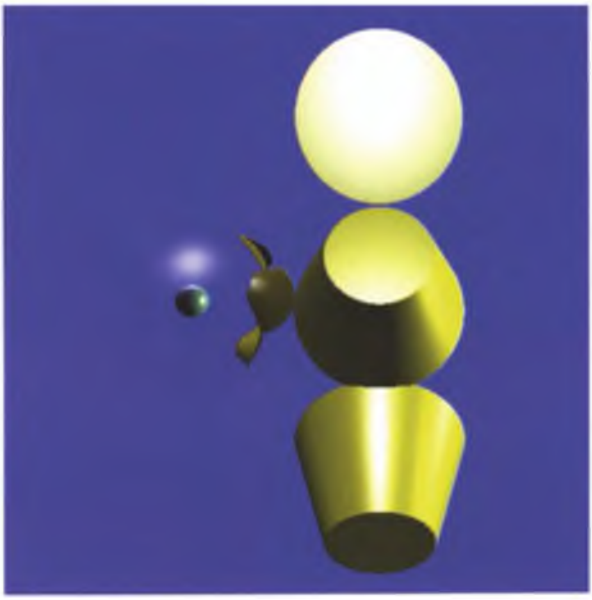


Abb. XIX Raytracing-Kombination. Die drei Kegelstümpfe sind diesmal mit einem festen Deckel ausgestattet. Dies ist durch eine Option im Raytracer möglich. Links daneben befindet sich eine Spiegelkugel, mit der die Szene auch von der Seite betrachtet werden kann. Die Szene wird durch zwei Lichtquellen beleuchtet.

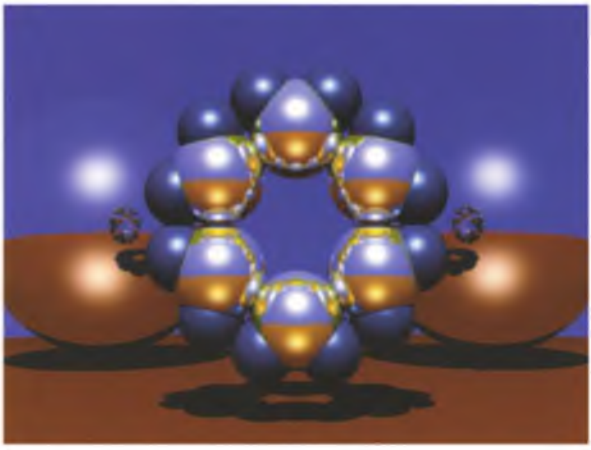


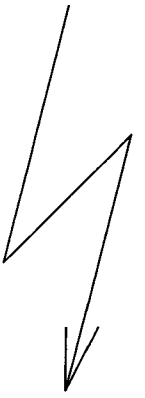
Abb. XX Der Molekülring, diesmal mit Raytracing berechnet. Mit dem beiliegenden Programm lassen sich auch komplexere Szenen darstellen.

Abb. XXI Das Bild wurde mit Supersampling durch den PD-Raytracer PV-RAY auf dem i860 in 3 Stunden und 3 Minuten berechnet. Dafür wurden bei 640 auf 480 Punkten allein 148 Millionen Schnitte mit dem verspiegelten Torus berechnet. Das Raytracing-Paket ist extrem umfangreich und mit Sourcecode in C erhältlich. Es besitzt eine komplexe Eingabesprache mit vielen Objekttypen, Shadern und Optionen.

V

Hardware

Rolf-Dieter Klein



20. Grafik-Hardware

In diesem Kapitel wollen wir einen kleinen Ausflug in die Hardwarewelt unternehmen und zeigen, wie Grafikkarten prinzipiell aufgebaut sind. Am Beispiel von gebräuchlichen PC-Karten werden auch einige spezielle Informationen vermittelt.

Die Grundlagen gelten jedoch auch für andere Rechner, wobei gerade beim PC aus historischen Gründen vieles komplizierter ist, als es sein müßte. Wenn Sie keinen PC besitzen, wird Ihnen dieses Kapitel dennoch wertvolle Kenntnisse über den Aufbau von Grafik-Hardware geben.

20.1 Aufbau und Wirkungsweise

20.1.1 Der Bildwiederholspeicher

Jeder Punkt, der auf dem Bildschirm sichtbar werden soll, steht im Speicher des Computers. Diesen Speicher nennt man Bildwiederholspeicher (frame buffer). Er befindet sich im Falle des PC im allgemeinen auf einer Grafikkarte. Die Größe dieses Speichers entscheidet auch über die maximale Auflösung des Bildschirms. Je größer dieser Speicher ist, desto mehr Bildpunkte können in ihm abgelegt werden.

Ein zweites wichtiges Kriterium ist die Anzahl der Bits, die für einen Bildpunkt verwendet werden. Bei einem Bit pro Pixel kann nur eine sogenannte monochrome Darstellung erreicht werden, denn im Speicher kann dann nur „Punkt an“ oder „Punkt aus“ gespeichert sein. Werden mehr Bits dafür bereitgestellt, so ist es möglich, mehrere Farben oder Graustufen pro Bildpunkt abzuspeichern. Für eine Auflösung von 640×480 Bildpunkten mit 8 Bit pro Bildpunkt wird also ein Bildwiederholspeicher mit 307200 Bytes benötigt. Da es keinen 307200 Byte großen Speicher gibt, wird die nächstgrößere Zweierpotenz verwendet, also 512 KByte (1K = 1024). Für eine monochrome Darstellung würden nur 38400 Bytes benötigt werden.

Um zu verstehen, was eine Bildschirmsteuerung mit dem Bildwiederholspeicher tun muß, schauen wir uns einmal die Arbeitsweise eines Bildschirms an. Abb. 20.1 zeigt einen Bildschirm beim Bildaufbau.

Der Elektronenstrahl der Bildröhre läuft dabei von links nach rechts und füllt den Bildschirm von oben nach unten. Am Ende einer jeden Zeile muß der

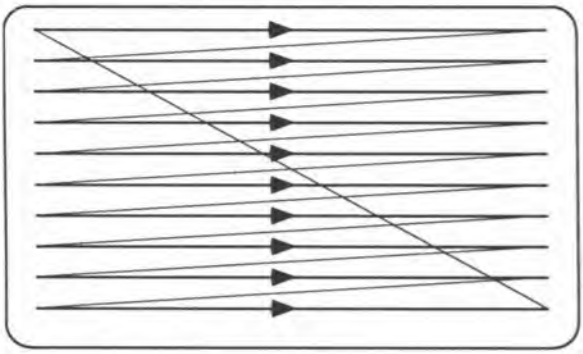


Abb. 20.1 Strahlverlauf bei einem Bildschirm

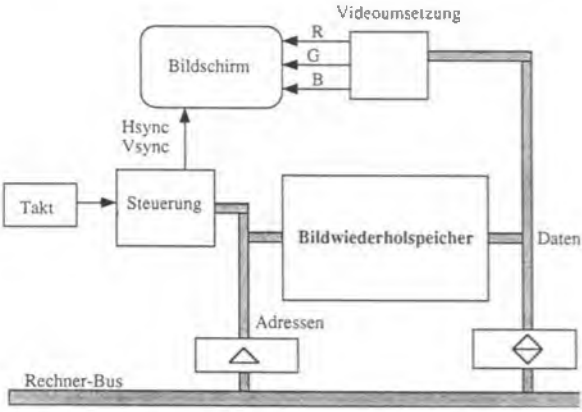


Abb. 20.2 Aufbau eines Bildwiederholspeichers

Strahl unsichtbar von rechts nach links zurückspringen und auf den Anfang der nächsten Zeile positioniert werden. Ist der Strahl ganz rechts unten angekommen, so springt er wieder nach links oben. Dieser Zeilenrücklauf ist ebenfalls unsichtbar und wird durch Dunkeltasten des Schreibstrahls erreicht. Der Bildschirm wird mit Hilfe zweier Synchronimpulse mit der Grafikkarte synchronisiert. Es gibt eine Synchronisation für die horizontale Ablenkung (H-Sync) und eine für die vertikale Ablenkung (V-Sync).

Am Anfang jeder Bildzeile wird dazu von der Grafik-Hardware ein H-Sync-Signal erzeugt und zu Beginn des kompletten Bildes ein V-Sync-Signal. Während der Strahl über den Bildschirm wandert, muß von der Steuerlogik synchron zur Bildposition der Bildinhalt ausgegeben werden. Dazu wird der Inhalt aus dem Bildwiederholspeicher gelesen und einem Videoumsetzer zugeführt. Abb. 20.2 zeigt ein Blockdiagramm.

Beim PC kann dieser Bildwiederholspeicher auch direkt vom Prozessor angesprochen werden. Die Steuerung muß dafür sorgen, daß der Bildwiederholspei-

cher immer zyklisch ausgelesen wird, und daß bei einem Zugriff vom PC aus dessen Daten in den Bildwiederholspeicher geschrieben werden können, ohne daß eine Kollision stattfindet. Bei den ganz alten CGA-Karten des PC sah man bei einem Zugriff durch den PC „Schnee" auf dem Bildschirm. Die Steuerung schaltet einfach den Speicher um und übergibt dem PC die Kontrolle. Dadurch wird zum Zeitpunkt des Zugriffs auf dem Bildschirm nur der gerade anliegende Wert des PC angezeigt und nicht der Wert, der hätte ausgelesen werden müssen.

Heute gibt es verschiedene Methoden, um Kollisionen zu vermeiden. Die einfachste ist, Zugriffe nur am Zeilen- und Bildschirmende zuzulassen, also genau dann, wenn der Strahl zurückläuft und ohnehin nichts auf dem Bildschirm geschrieben werden darf. Der Prozessor muß dann aber immer warten, bis ein solcher erlaubter Zeitpunkt erreicht ist. Dies ist bei normalen Videofrequenzen etwa alle $64\mu s$ der Fall.

Besser ist es, den Zugriff zu verzahnen. Dann können Prozessor und Bildschirm-Controller abwechselnd auf den Bildwiederholspeicher zugreifen. Dazu müssen die verwendeten Speicher aber doppelt so schnell sein wie beim einfachen Zugriff, was bei hohen Bildschirmauflösungen kritisch sein kann. Ein anderer Weg ist es, spezielle Speicherbausteine zu verwenden.

Bei den sogenannten VRAM (Video Random Access Memory) gibt es die Möglichkeit, den Speicher über einen Extra-Ausgang seriell auszulesen und gleichzeitig vom PC aus auf die Speicherzellen zuzugreifen. Dazu werden bei diesen Speichern per Impuls z.B. 512 Bits in ein internes Schieberegister kopiert. Das Schieberegister besitzt einen eigenen Ausgang und kann dann seriell, also Bit für Bit ausgelesen werden. Der PC wird nur noch während des Kopierens für ganz kurze Zeit (ca. 200 ns) blockiert. Das Kopieren geschieht immer nur dann, wenn das Schieberegister leer ist. Dadurch kann der Prozessor fast 100% der Zeit im Speicher verbringen und dort den Bildaufbau übernehmen, ohne dabei von der Bildschirmsteuerung gestört zu werden.

20.1.2 Die Farbpalette

Auf allen modernen Grafikkarten, insbesondere auf VGA-Karten, findet sich eine sogenannte Farbpalette. Was hat es damit auf sich?

In der frühen IBM-Computer-Zeit, wo die ersten Grafikkarten auftauchten, trat das Problem der Farbzuordnung zum ersten Mal auf. Die CGA-Karten (Color Graphics Adapter) konnten im Grafik-Modus nur vier verschiedene Farben auf einmal darstellen, da aus Speichermangel nur zwei Bits für die Farbzuordnung vorhanden waren, und damit nur vier Kombinationsmöglichkeiten. Auf diesen Karten gab es bereits eine einfache Form der Farbpalette. Mit einem Befehl (`INT10, AH = $0B`) konnte man zwischen zwei fest eingestellten Farbzuordnungen umstellen und zwar zwischen

1. Schwarz, Rot, Grün und Braun und
2. Schwarz, Cyan, Magenta und Weiß.

Tabelle 20.1 Mögliche EGA-Farben

Index	Bitkombination	Farbe am Bildschirm
0	0 0 0 0 0 0	Schwarz
1	0 0 0 0 0 1	Blau
2	0 0 0 0 1 0	Grün
3	0 0 0 0 1 1	Türkis
4	0 0 0 1 0 0	Rot
5	0 0 0 1 0 1	Violett
6	0 1 0 1 0 0	Braun
7	0 0 0 1 1 1	Grau
8	1 1 1 0 0 0	Dunkelgrau
9	1 1 1 0 0 1	Hellblau
10	1 1 1 0 1 0	Hellgrün
11	1 1 1 0 1 1	Helltürkis
12	1 1 1 1 0 0	Hellrot
13	1 1 1 1 0 1	Hellviolett
14	1 1 1 1 1 0	Gelb
15	1 1 1 1 1 1	Weiß

Gebessert hat sich die Situation mit dem Aufkommen von EGA-Karten (Enhanced Graphics Adapter). Im Grafik-Modus lassen sich hier 16 verschiedene Farben in einem Bild gleichzeitig darstellen, denn es stehen 4 Bits pro Bildpunkt für die Farbauswahl zur Verfügung. Jeder der 16 Kombinationsmöglichkeiten läßt sich nun eine Farbe zuordnen. Dabei können 64 verschiedene Farbabstufungen verwendet werden. Die folgende Abbildung zeigt die Zuordnung.

R-Intens	G-Intens	B-Intens	R	G	B

Die EGA-Palette ist standardmäßig mit einer fertigen Zuordnung initialisiert. Tabelle 20.1 zeigt diese Farbwerte. Die Farbtabelle wird in einem RAM-Speicher auf der EGA-Karte abgelegt.

Bei einem Zugriff auf den Bildwiederholspeicher wird der Code jedes Pixels über die Palette geleitet und dann erst als Farbwert ausgegeben. Abb. 20.3 zeigt dazu das Schema. Der 4-Bit-Code des Bildwiederholspeichers wählt eine der 16 Speicherzeilen aus. In einer Speicherzeile sind 6 Bit des Farbcodes abgelegt. Dieser wird nun an einen Digital-Analog-Umsetzer (D/A-Umsetzer) geführt, der hier symbolisch als Widerstandsnetz eingezeichnet ist. Jeweils zwei Bits der Palette bestimmen die Farbe einer der Farbkomponenten Rot, Grün oder Blau.

Das Schema ist zwar so schon flexibler, jedoch gibt es nicht mehr als 64 verschiedene Farben in der Palette. Pro Farbkomponente existieren sogar nur vier Möglichkeiten. Eingezeichnet ist in diesem Schema auch der Zugang vom Prozessor, denn dieser muß die Farbpalette auch beschreiben können. Damit dies nicht zu Störungen führt, schreibt man die Daten für die Farbpalette in der vertikalen Austastlücke des Bildes hinein. Das BIOS im PC wartet automatisch auf die Austastlücke, wenn man Paletteneinträge ändert. Wenn Sie die

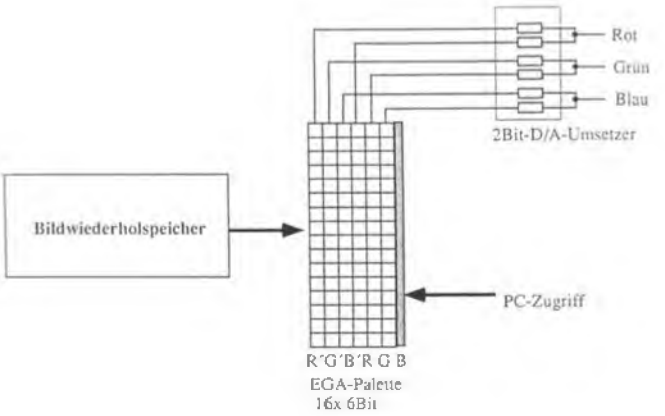


Abb. 20.3 Blockdiagramm der EGA-Farbpalette

Palette mit einem solchen BIOS-Aufruf ändern, so benötigt jede Änderung ca. 16 Millisekunden.

Bei VGA-Karten (Video Graphics Array) wurde dieses Schema weiter ausgebaut. Nun ist es möglich, bis zu 8 Bit pro Pixel zu verwenden, also 256 Farben gleichzeitig auszugeben. Die Farbpalette wurde ebenfalls stark erweitert. Dort stehen pro Farbkomponente 6 Bit zur Verfügung. Damit kann für jede der 256 Kombinationsmöglichkeiten pro Pixel aus 262144 Farben (64 Rotschattierungen * 64 Grünschattierungen * 64 Blauschattierungen) ausgewählt werden. Die Palette umfaßt damit 256 Einträge mit je 18 Bit. In Abb. 20.4 ist dazu ein Blockdiagramm dargestellt.

Der Bildwiederholspeicher liefert einen Datenstrom mit je 8 Bit pro Pixel. Dieser gelangt an die Farbpalette und wird dort als Adresse verwendet. Dadurch wird eine der 256 Zeilen ausgewählt. Da eine Zeile aus 18 Bit besteht, werden somit 18 Bit ausgelesen. Diese gelangen in Gruppen zu je 6 Bit an 3 D/A-Umsetzer, die aus einem digitalen Wert zwischen Null und 63 einen analogen Wert, d.h. eine dazu proportionale Spannung generieren.

Der analoge Wert gelangt dann als Rot-, Grün- oder Blau-Signal direkt an den Bildschirm. Die Helligkeit des dargestellten Punktes entspricht der Spannung. Je höher die Spannung, desto heller ist der Punkt. Der Prozessor kann über einen eigenen Eingang den Inhalt der Farbpalette ändern. Dabei wird jedoch der Auslesevorgang für den Bildwiederholspeicher blockiert und es würden Störungen auf dem Bildschirm erscheinen. Daher darf auch die Farbpalette nur dann geändert werden, wenn sich der Schreibstrahl des Bildschirms gerade in der Austastlücke (Rücklauf vom unteren Bildschirmrand zum oberen) befindet. Dies wird auch durch das BIOS garantiert.

Neben diesen 18-Bit-Paletten gibt es auch Bausteine mit 24 Bit, bei denen man aus 16777216 Farbschattierungen auswählen kann. Sie sind jedoch auf normalen VGA-Karten nicht vorhanden. Mit diesen Paletten lassen sich noch feinere Farbabstufungen darstellen. Will man mit der VGA-Palette eine Grau-

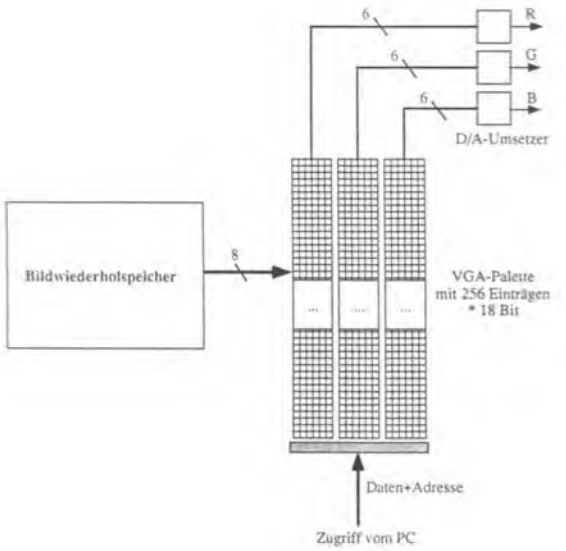


Abb. 20.4 Blockdiagramm der VGA-Farbpalette

skala aufbauen, so erhält man 64 Stufen. Diese einzelnen Stufen sind durchaus gut erkennbar. Bei 256 Stufen würde man den Verlauf nicht mehr sehen.

Zum Laden der Farbpalette gibt es verschiedene Möglichkeiten. Man kann die Turbo-Pascal Prozedur `SetRGBPalette(index,r,g,b)` oder den BIOS-Interrupt $10 verwenden – wie im folgenden Beispiel. Beim BIOS-Aufruf gibt es zwei Möglichkeiten: Man kann die Paletteneinträge einzeln laden oder gleich eine ganze Tabelle als Parameter angeben. Letzteres hat den Vorteil, daß die Palette schneller geladen wird. Beim einzelnen Laden wartet das BIOS immer auf die Austastlücke und bei 256 Einträgen benötigt man ca. 4 Sekunden (16ms $\times$ 256) zum Umladen der gesamten Palette. Nun die beiden BIOS-Funktionen im Detail.

20.1.2.1 Setzen eines einzelnen Farbregisters

INT $10	Interrupt $10
AH = $10	Palettenfunktionen
AL = $10	Befehl Setze einzelnes Farbregister
BX = 0...255	Farbregister
CH = 0...63	Grünanteil
CL = 0...63	Blauanteil
DH = 0...63	Rotanteil

In Turbo-Pascal kann man in diesem Fall auch `SetRGBPalette` verwenden.

```
USES dos;

PROCEDURE mySetRGBPalette(index ,r,g,b : WORD);
VAR
  regs : Registers;
BEGIN
  regs.ah := $10;
  regs.al := $10;
  regs.bx := index;
  regs.dh := r;
  regs.cl := b;
  regs.ch := g;
  intr($10,regs);
END;
```

20.1.2.2 Setzen eines Blocks von Farbregistern

INT $10	Interrupt $10
AH = $10	Palettenfunktionen
AL = $12	Befehl Block setzen
BX = 0...255	Adresse des ersten Farbregisters
CX = 1...255	Anzahl der zu setzenden Farbregister
ES:DX	Adresse des Datenblocks mit den Farbwerten. Dort sind als Bytefolge R,G,B,R,G,B... gespeichert.

Der Blockbefehl arbeitet viel schneller, denn er kann die gesamte Palette auf einmal neu setzen. Dabei wartet auch dieser Befehl auf die Austastlücke, aber eben nur ein Mal.

In Pascal geschieht die Programmierung wie folgt:

```
USES dos;
TYPE
  farbtype = ARRAY[0..767] OF BYTE; (*  3*256  *)

PROCEDURE mySetblockRGB(feld : farbtype);
VAR
  regs : Registers;
BEGIN
  regs.ah := $10;
  regs.al := $12;
  regs.bx := 0;          (* hier von Index 0 an *)
  regs.cx := 256;        (* Alle Register *)
  regs.es := seg(feld); (* Segmentadresse des Felds *)
  regs.dx := ofs(feld); (* Offsetadresse des Felds *)
  intr($10,regs);
END;
```

Abb. 20.5 Paletten-Animation mit einem Band

20.1.2.3 Programmbeispiel

Auf der beigefügten Diskette finden Sie das Programm `PALETTE.PAS`. Es gibt zunächst die voreingestellte Palette auf dem Bildschirm aus. Wenn Sie dann die Leertaste drücken, so wird die Palette mit neuen Werten überschrieben. Dabei wird der Einzel-Setz-Befehl verwendet. Sie können daher das Umladen auf dem Bildschirm beobachten. Anschließend wird die Palette mit Hilfe des Blockbefehls umprogrammiert. Dabei wird sie jeweils mit einem anderen Inhalt belegt, und zwar so, daß der alte Eintrag mit Index i dem neuen Eintrag mit Index $i-1$ entspricht.

Damit sind wir schon beim nächsten Thema, der Farbpalettenanimation. Der Hintergrund hat ebenfalls seine Farbe gewechselt, dies kommt daher, daß der Eintrag 0 als Hintergrundfarbe verwendet wird. Es gibt bei VGA-Karten noch ein weiteres Farbregister, das sogenannte Overscan-Register. Dies bestimmt die Farbe für den Bildschirmrand.

20.1.3 Animation mit Farbpaletten

Wenn man die Palette während des Programmlaufs umprogrammiert, so ergibt sich sofort eine Änderung im Bild. Wie Sie schon im letzten Beispiel gesehen haben, kann man damit auch Animationseffekte erreichen. Es entsteht der Eindruck, als ob die Rechtecke im Bild durchlaufen würden. Dieser Effekt läßt sich auch noch weiter ausbauen. Abb. 20.5 zeigt ein Beispiel. Dargestellt ist ein weißes Band, auf dem sich rote Farbflächen von links nach rechts bewegen sollen. Man könnte nun per Programm das Band für die einzelnen Bewegungsphasen immer neu zeichnen, doch bei komplizierteren Formen ist das von der Rechenzeit her gar nicht machbar. Mit der Farbpalette geht es sehr einfach.

Dazu muß man das Band in kleinere Segmente zerlegen und mit unterschiedlichen Farbcodes belegen. Abb. 20.6 zeigt die Lösung. Den Segmenten wird abwechselnd der Farbcode 1, 2 und 3 zugewiesen. In der Farbpalette speichert man im ersten Zeitschritt für den Code 1 den Farbwert für Rot, für den Code 2 den Farbwert für Weiß und für den Code 3 ebenfalls den Farbwert für Weiß. Für den Zeitschritt 2 wird dann die Palette umprogrammiert. Dabei wird Code 1 der Farbwert Weiß zugeordnet und Code 2 der Farbwert Rot. Code 3 wird auf Weiß belassen. Damit wandert das rote Farbband scheinbar einen Schritt nach rechts.

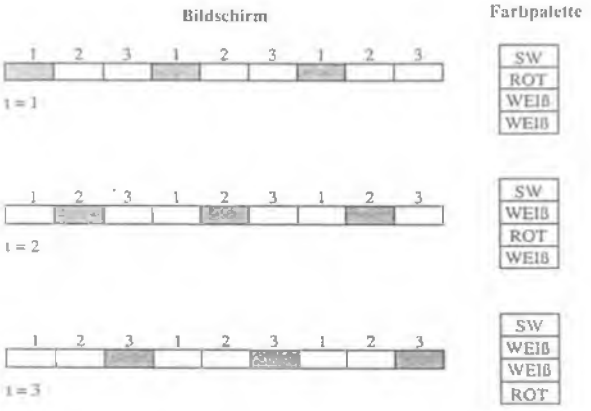


Abb. 20.6 Ablauf der Palettenanimation

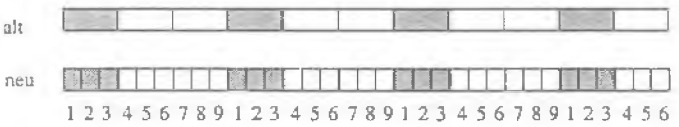


Abb. 20.7 Unterteilung der farbigen Blöcke

Wird nun im letzten Schritt Code 1 mit Weiß, Code 2 mit Weiß und Code 3 mit Rot belegt, so wandert der Balken wieder nach rechts. Danach kann man wieder von vorne beginnen und Code 1 mit Rot, Code 2 mit Weiß und Code 3 mit Weiß belegen usw. Der Balken bewegt sich nun dauernd. Auf der Diskette ist das Programm unter dem Namen `PALANIM.PAS` abgelegt. Dabei werden zwei Balken dargestellt. Beim oberen Balken sind die Felder 8 Pixel breit und beim unteren nur zwei Pixel.

Die Animation wirkt hier noch sehr grob, da es nur drei Phasen für die Bewegung gibt. Eine Unterteilung in mehr Phasen bringt auf jeden Fall eine Verbesserung. In Abb. 20.7 werden für das rote Feld drei Farbeinträge verwendet und insgesamt neun Einträge für die ganze Animation. Zur Festlegung der Anzahl von Feldern, die man in der Farbtabelle verwendet, muß man eigentlich auch die Blockgröße der zu bewegenden Elemente berücksichtigen.

In unserem Fall hatten wir 8 Pixel für einen Block (x-Richtung) verwendet bzw. zwei Pixel beim darunterliegenden Block (siehe Programm `PALANIM.PAS`). Für den Block mit 8 Pixeln verwenden wir also vier Einträge und damit insgesamt 12 Palettenadressen. Die kleinen Blöcke mit zwei Pixel Breite können maximal aus zwei Farbwerten zusammengesetzt werden. Bei der Darstellung spielt auch die gewünschte Animationsgeschwindigkeit eine wichtige Rolle.

Im Programm `PALANIM.PAS` ist auch die neue Variante enthalten und wird verwendet, wenn man die Leertaste drückt. Auch diesmal werden zwei Balken

dargestellt. Der untere Balken wird dabei mit eine Pixelbreite von 2 gezeichnet. Jedes Pixel erhält die Farbe `COLOR+2`. Damit ist die Abstufung der Animation nicht so fein, dafür wird jedoch die Durchlaufgeschwindigkeit erhöht. Sie können es einmal mit Werten von 1 und 4 versuchen. Bei 4 wird ein Flimmereffekt entstehen, da nicht zu jedem Zeitpunkt ein rotes Feld sichtbar ist.

Wenn man die Animationsgeschwindigkeit abschätzen will, so kann man sich der folgenden Formel bedienen:

$$v = \frac{\text{Monitorfrequenz} \times \text{Feldbreite}}{\text{Farbanzahl}} \left[\frac{\text{Pixel}}{\text{sec}}\right]$$

Dabei ist „Monitorfrequenz" die vertikale Ablenkfrequenz, die normalerweise zwischen 60 und 75 Hz liegt. „Feldbreite" ist die Breite einer Farbfläche in Pixel und „Farbanzahl" die Zahl der Paletteneinträge, die pro Farbe verwendet wurden. Für das zweite Beispiel ergibt sich damit

$$v = \frac{60 \cdot 8}{4} \left[\frac{\text{Pixel}}{\text{sec}}\right] = 120 \left[\frac{\text{Pixel}}{\text{sec}}\right]$$

Das rote Feld bewegt sich also in 2.6 Sekunden vom linken zum rechten Rand des Bildschirms, wenn dieser 320 Punkte breit ist.

20.1.4 Nachbildung von True-Color

Wenn man nun mehr als 256 Farben braucht, aber nur eine normale VGA-Karte besitzt, so kann man mit einem Trick dennoch scheinbar mehr Farben herbeizaubern. Man verwendet abwechselnd verschiedene Farben nebeneinander auf dem Bildschirm. Im Auge mischen sich dann die Farben. Die Mischfarbe ist die gewünschte fehlende Farbe. Allerdings wird dadurch das Bild unschärfer, denn man verliert Pixel für die Auflösung. Es ist dann vorteilhaft, eine Karte mit hoher Auflösung zu besitzen, damit das Auge nebeneinanderliegende Pixel nicht mehr einzeln wahrnimmt und der Mischeffekt auch erhalten bleibt, wenn man näher an den Bildschirm herangeht.

20.1.4.1 Halbtontechnik

Beim Zeitungsdruck stehen für Schwarzweißfotos nur zwei Farben zur Verfügung, entweder ist Druckerfarbe da oder nicht. Um trotzdem Graustufen darstellen zu können, bedient man sich eines einfachen Tricks. Die Druckerschwärze wird als feines Punktraster aufgetragen, wobei die Punkte umso größer sind, je dunkler sie sein sollen. Dieses Verfahren nennt man „Halbtonverfahren" oder auf englisch „halftone" .

Dieses Verfahren wird zum Beispiel auch bei Laserdruckern verwendet, um Fotos auszugeben. Der Laserdrucker kann eigentlich auch nur zwischen Punkt und Nicht-Punkt unterscheiden. Die Punktgröße ist dabei normalerweise immer gleich groß. Die unterschiedliche Punktgröße wird dann durch Zusammenfassen von mehreren kleinen Punkten erreicht. Abb. 20.8 zeigt ein Foto, das auf einem

Abb. 20.8 Halftone-Technik

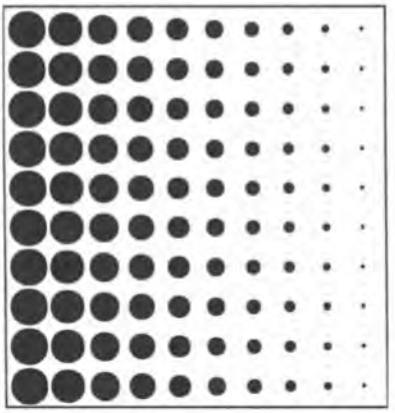


Abb. 20.9 Halftone-Pixel

Laserdrucker ausgegeben wurde. Es ist stark vergrößert, so daß Sie die einzelnen Punkte deutlich erkennen können. Eine Grautreppe ist in Abb. 20.9 abgebildet, wobei ideale Punkte verwendet wurden.

Will man solche Punkte mit einem Laserdrucker ausgeben, so muß man die Punkte aus kleineren Punkten zusammensetzen. Ein Laserdrucker druckt z.B. mit einer Auflösung von 300 dpi (Punkte pro Inch), ein Punkt ist also 0.085 mm groß. Es gibt allerdings auch Laserdrucker, die unterschiedliche große Punkte direkt drucken können. Doch auch diese Punkte haben nur einen begrenzten Bereich, innerhalb dessen der Durchmesser variiert werden kann und so findet auch dort eine Zerlegung statt.

Wenn die Punkte jedoch in ein Raster zerlegt werden, so nimmt die nutzbare Auflösung weiter ab. Abb. 20.10 zeigt den Vorgang. Die idealen Punkte werden in ein Raster gepreßt. Dadurch entstehen nur noch grob wirkende Punkte. Der größte Punkt bestimmt dann die darstellbare Auflösung. Besteht der größte Punkt aus einem 10×10 Feld und hat ein solches Feld eine Auflösung von 300

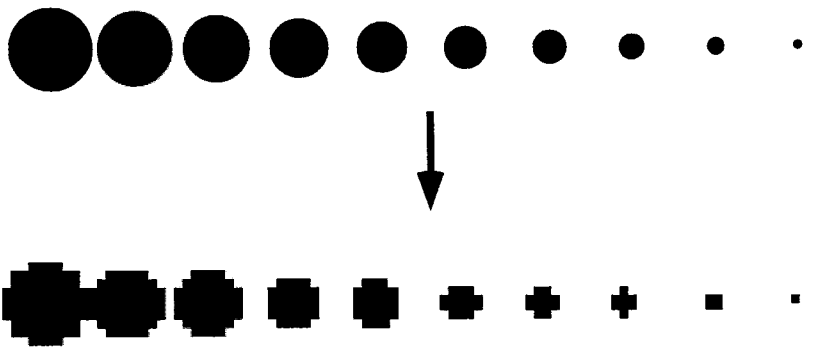


Abb. 20.10 Die Umsetzung der Halftone-Pixel in ein Punktraster

dpi, so haben die damit dargestellten Bilder nur noch 30 dpi Auflösung. Für eine Bildschirmdarstellung ist dieses Verfahren nicht geeignet. Dort hat man bei einer Standardauflösung von 640×480 Punkten ca. 72 dpi Auflösung. Bei 10×10 Punkten für einen Halbtonpunkt wäre die Darstellung auf 7.2 dpi begrenzt und das sind umgerechnet nur noch 2.83 Punkte pro Zentimeter (1 inch = 2.54cm).

Zum Glück gibt es Verfahren, die für eine Bildschirmdarstellung besser geeignet sind als das Halbtonverfahren. Sie werden in den nächsten Abschnitten beschrieben.

20.1.4.2 Ordered Dither nach Bayer

Hier soll eine Technik vorgestellt werden, die sich für eine Bildschirmdarstellung sehr gut eignet. Die Punkte eines Laserdruckers können nicht so ohne weiteres isoliert gedruckt werden, da dann oft störende Effekte wie z.B. Moiré-Muster auftreten. Daher wird man bei Laserdruckern immer zu Cluster-Verfahren, wie sie im vorherigen Abschnitt beschrieben wurden, greifen.

Das Dithering hat immer die Aufgabe, nicht vorhandene Farbabstufungen durch Mischeffekte zu erzeugen. Wenn man also bei einer Grafikkarte nicht soviele Farben gleichzeitig darstellen kann, wie man braucht, so greift man zu einem Dithering-Verfahren. In diesem Abschnitt soll ein Verfahren dargestellt werden, das sich in der Praxis sehr bewährt hat. Das Dithering-Verfahren nach Bayer arbeitet mit einem regulären Punktraster, das mit Hilfe einer gespeicherten Matrix bestimmt wird. Sie legt fest, ob ein Bildschirmpunkt dunkel oder hell erscheinen soll. Mit diesem Verfahren ist es möglich, auf einem reinen Schwarzweiß-Grafikschirm Graustufen darzustellen. Hat man Farben oder mehr Abstufungen zur Verfügung, so kann das Verfahren diese zur Darstellung ausnutzen.

Der Vorteil bei diesem Verfahren ist, daß es auch verwendet werden kann, wenn die Auflösung des Bildes gleich der des Bildschirms ist. Beim Halbtonverfahren mußte das Bild eine kleinere Auflösung haben als der Bildschirm bzw. Drucker.

Doch zunächst einmal die Dithermatrix für 2×2 Punkte.

$$D = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

Mit dieser Dithermatrix lassen sich 5 Graustufen codieren. Der Algorithmus arbeitet dann wie folgt:

```
PROCEDURE SetzeDitherPixel(x,y,grey5 : INTEGER);
BEGIN
  IF (D[x MOD 2][y MOD 2] < grey5) THEN setdot(x,y);
END;
```

Dabei hat `grey5` einen Wertebereich von $0 \ldots 4$. Je größer der Wert von `grey5` ist, desto mehr Punkte werden gesetzt. Ist `grey5` gleich Null, so wird nie ein Wert gesetzt. Bei `grey5` gleich 4 werden alle Punkte gesetzt. Hier sind nur 5 Graustufen unterscheidbar, jedoch kann die Dithermatrix auch für größere Bereiche definiert werden, z.B. für ein 4×4-Punktraster. Um für einen Punkt zu bestimmen, ob er hell oder dunkel sein soll, wird zunächst eine Modulo-Operation mit der Koordinate des Punktes durchgeführt. Dazu ein Beispiel:

Der Punkt bei $x = 123, y = 328$ soll mit der Graustufe 3 dargestellt werden. Zunächst muß für x und y der Index in die Dithermatrix bestimmt werden. Da die Dithermatrix 2×2 Elemente groß ist, wird `x mod 2` gebildet. Man erhält für `x mod 2` den Wert $(123 \bmod 2) = 1$ und für `y mod 2` liefert $(328 \bmod 2)$ den Wert 0. Nun muß also bei `D[1][0]` nachgesehen werden. Dort steht der Wert 2. Der Wert 2 ist kleiner als die gewünschte Graustufe 3 und damit wird der Punkt auf dem Bildschirm gesetzt.

So kann man für jeden Punkt, den man zeichnen will, bestimmen, ob er gesetzt wird oder nicht. Durch dieses Verfahren ergibt sich das gerasterte Graustufenbild. Dabei entscheidet nicht der einzelne Punkt, sondern die gebildeten Punktmuster über einer Fläche. Nehmen wir eine 4×4-Matrix.

$$D = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

Das Programm kann nun 17 Graustufen unterscheiden und darstellen:

```
PROCEDURE SetzeDitherPixel(x,y,grey17 : INTEGER);
BEGIN
  IF (D[x MOD 4][y MOD 4] < grey17) THEN setdot(x,y);
END;
```

Das Besondere an dem Verfahren ist, daß für jedes Pixel auf dem Bildschirm die Entscheidung getroffen wird, ob es an oder aus ist, unabhängig vom Nachbarn, aber abhängig von der Graustufe.

Abb. 20.11 zeigt die Darstellung des schon früher verwendeten Fotos nach diesem Verfahren. Obwohl nur recht grobe Bildpunkte verwendet wurden, lassen

Abb. 20.11 Ordered Dither nach Bayer

sich auch Graustufen noch gut erkennen.

Die Dithermatrix läßt sich auch für noch mehr Graustufen aufstellen, hier für 65 Graustufen mit einer 8×8-Matrix.

$$D = \begin{pmatrix} 0 & 32 & 8 & 40 & 2 & 34 & 10 & 42 \\ 48 & 16 & 56 & 24 & 50 & 18 & 58 & 26 \\ 12 & 44 & 4 & 36 & 14 & 46 & 6 & 38 \\ 60 & 28 & 52 & 20 & 62 & 30 & 54 & 22 \\ 3 & 35 & 11 & 43 & 1 & 33 & 9 & 41 \\ 51 & 19 & 59 & 27 & 49 & 17 & 57 & 25 \\ 15 & 47 & 7 & 39 & 13 & 45 & 5 & 37 \\ 63 & 31 & 55 & 23 & 61 & 29 & 53 & 21 \end{pmatrix}$$

Die Dithermatrix läßt sich auch rekursiv aus einer 2×2-Dithermatrix berechnen. Dabei hat die Matrix immer eine Zweierpotenz als Dimension. Ausgegangen wird von der 2×2-Dithermatrix. Daraus läßt sich eine 4×4-Dithermatrix ableiten, daraus die 8×8-Matrix usw.

$$D^{2n} = \begin{pmatrix} 4D^n & 4D^n + 2E^n \\ 4D^n + 3E^n & 4D^n + E^n \end{pmatrix}$$

Dabei ist E eine Matrix, die komplett mit dem Wert 1 gefüllt ist, z.B.:

$$E^4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}.$$

Das Dithering-Verfahren kann so ausgedehnt werden, daß auch mehrere Farben und schon vorhandene Graustufen dargestellt werden können. Wir wollen z.B. eine 24-Bit-Farbgrafik auf einer Grafikkarte darstellen, die nur eine 8-Bit-Auflösung hat. Dabei sind bei der 24-Bit-Farbgrafik jeweils 8 Bit pro Grundfarbe (Rot, Grün und Blau) vorhanden.

Um dies zu realisieren, müssen die 8 Bit der Grafikkarte erst einmal auf die Grundfarben aufgeteilt werden. Dann kann das Dithering-Verfahren für jede Grundfarbe getrennt durchgeführt werden. Dazu werden üblicherweise 3 Bit für Rot, 3 Bit für Grün und 2 Bit für Blau reserviert. Auf dem Monitor hat Blau also nur 3 Farben (Schwarz, Dunkelblau, Blau, Hellblau). Für die anderen Grundfarben reserviert man mehr Schattierungen.

Warum gerade für Blau so wenig? Bei Blau kann das menschliche Auge nicht so viele Schattierungen unterscheiden, wie bei Rot oder Grün. Daher genügt eine geringere Farbauflösung. Bei einer 16-Bit-Farbkarte würde man entsprechend 6 Bit für Rot, 6 Bit für Grün und 4 Bit für Blau verwenden.

Die verschiedenen Schattierungen müssen nun in die Farbpalette der Grafikkarte programmiert werden. Dies geschieht z.B. mit dem folgenden Programm (auf der Diskette im Beispiel `DITHER1.PAS`).

```
PROCEDURE initpalette;   (* Farbtabelle aufbauen *)
VAR
  i:INTEGER;
  r,g,b : INTEGER;
  regs : Registers;
BEGIN
  FOR i:=0 TO 255 DO BEGIN
    r := (i AND 7) * 9;
    g := ((i SHR 3) AND 7) * 9;
    b := ((i SHR 6) AND 3) * 21;
    setrgbpalette(i,r,g,b);
  END;
END;
```

Wir möchten nun auf dem Bildschirm einen Farbverlauf darstellen. Dazu soll es die Routine `setdot(x,y,r,g,b)` geben mit `x,y` als Integer und `r,g,b` als `DOUBLE` für den Farbwert im Bereich 0.0...1.0. Das Programmstück sieht dann so aus:

```
FOR y:=0 TO yres-1 DO BEGIN
  FOR x:=0 TO xres-1 DO BEGIN
     r := x / (xres-1);
     g := y / (yres-1);
     b := 1 - r/2 - g/2;
     setdot(x,y,r,g,b);
  END;
END;
```

Dadurch muß bei $x = 0, y = 0$ als links oben die Farbe Blau vorhanden sein, rechts oben Rot und links unten Grün. Dazwischen ergeben sich Mischfarben.

Zunächst wollen wir `setdot` einmal ganz ohne Dithering entwerfen, nur unter Verwendung der Schattierung der Grundfarben, wie sie in der Farbpalette mit `initpalette` definiert wurden.

```
PROCEDURE setdotohnedither(x,y: INTEGER; rcol, gcol, bcol: DOUBLE);
VAR
  di : INTEGER;
  colorindex : INTEGER;
  r1, g1, b1 : INTEGER;
BEGIN
  r1 := trunc(rcol * 7.0);
  g1 := trunc(gcol * 7.0);
  b1 := trunc(bcol * 3.0);
  colorindex := r1 + (g1 SHL 3) + (b1 SHL 6);
  moveto(x,y);
  setcolor(colorindex);
  lineto(x,y);
END;
```

Die Parameter `rcol`, `gcol` und `bcol` sind Gleitkommazahlen zwischen 0.0 und 1.0. Wenn der Wert 0 ist, so ist die entsprechende Farbkomponente dunkel, bei 1.0 ist die maximale Helligkeit zu nehmen.

Da Rot und Grün jeweils 7 verschiedene Schattierungen haben, wird `rcol` und `gcol` mit 7 multipliziert und das Ergebnis dann in eine Integerzahl gewandelt, die dann nur noch die Werte 0, 1, 2...7 annehmen kann. Bei der Blaukomponente wird entsprechend mit 3 multipliziert. Daraus wird nun der Index in die Farbpalette berechnet, indem die Komponenten jeweils auf die entsprechende Bitposition geschoben und dann addiert werden. Der Index hat damit einen Bereich von 0...255.

Achtung: Ein Überlauf der Farbkomponenten wird nicht berücksichtigt. Wenn der Wertebereich nicht von vorneherein auf das Intervall zwischen 0 und 1 begrenzt ist, sind entsprechende Abfragen in die Prozedur aufzunehmen. Im Programm auf der Diskette sind diese Abfragen eingebaut, auch wird dort gerundet und damit die Palette besser ausgenutzt.

Sie können den Farbverlauf mit dieser einfachen Routine ansehen, wenn Sie das Programm `DITHER1` starten.

Das Programm für eine `setpixel`-Implementierung mit Dithering sieht dann so aus:

```
PROCDURE setpixel(x, y : INTEGER; rcol, gcol, bcol : DOUBLE);
VAR
  di : INTEGER;
  colorindex : INTEGER;
  r, g, b, r1, g1, b1 : INTEGER;
BEGIN
  di := dith[x MOD 8][y MOD 8];
  r := trunc(rcol * 64.0 * 7.0);
  g := trunc(gcol * 64.0 * 7.0);
  b := trunc(bcol * 64.0 * 3.0);
  r1 := r SHR 6;
  g1 := g SHR 6;
  b1 := b SHR 6;
```

```
    IF ((r AND 63) > di) THEN r1 := r1 + 1;
    IF ((g AND 63) > di) THEN g1 := g1 + 1;
    IF ((b AND 63) > di) THEN b1 := b1 + 1;
    IF (r1 > 7) THEN r1 := 7;
    IF (g1 > 7) THEN g1 := 7;
    IF (b1 > 3) THEN b1 := 3;
    colorindex := r1 + (g1 SHL 3) + (b1 SHL 6);
    moveto(x,y);
    setcolor(colorindex);
    lineto(x,y);
  END;
```

Das Programm ähnelt der vorhergehenden Routine, nur wird hier zusätzlich die Dithermatrix verwendet. Außerdem werden die Farbkomponenten noch zusätzlich mit 64.0 multipliziert, der Anzahl der Graustufen, die durch eine 8×8-Dithermatrix erreicht werden können. Damit liegt `r` und `g` im Bereich 0...448 und `b` im Bereich 0...192. Daraus wird zunächst ein Basis-Index berechnet.

`r1`, `g1` und `b1` liegen jeweils im Bereich 0...7 bzw. 0...3. In der Variablen `di` steht der für diesen Punkt gültige Wert der Dithermatrix. Die Farbkomponente `r` wird in zwei Teile zerlegt, den Basis-Index `r1` und einem Rest `r AND 63`. Entsprechendes gilt bei den Komponenten `g` und `b`. Dieser Rest wird nun mit dem Ditherwert `di` verglichen. Ist er größer als `di`, dann wird der Wert Eins auf die Farbkomponente `r1`, `g1` oder `b1` addiert. Danach findet noch ein Überlauftest statt und dann wird der Farbindex berechnet, wie zuvor bei `setdotohnedither`.

Im Programm `DITHER1` ist auch diese Routine eingebaut. Nach der Ausgabe des Bildes ohne Dithermatrix wartet das Programm auf eine beliebige Taste. Drücken Sie irgendeine Taste, so wird das Bild mit der Dithervariante überschrieben. Hier läßt sich nun auch der Farbverlauf erkennen.

Störend beim „ordered dithering" ist das reguläre Raster, das man deutlich wahrnimmt, insbesondere wenn man Fotos damit darstellt. Dies zu vermeiden ist Ziel des nächsten Dithering-Verfahrens.

20.1.4.3 Floyd-Steinberg

Dieses Verfahren wird auch als Fehlerverteilungsverfahren bezeichnet oder auf Englisch: error diffusion. Es ist speziell geeignet, wenn die Auflösung des darzustellenden Bildes der Bildschirmauflösung entspricht.

Vorab das Resultat: In Abb. 20.12 ist unser Beispielfoto mit diesem Verfahren dargestellt. Im Gegensatz zur Abb. 20.11 scheinen mehr Details sichtbar zu werden. Auch sind keine störenden Raster erkennbar. Der Nachteil dieses Verfahrens ist, daß die Bildpunkte nicht mehr unabhängig voneinander dargestellt werden können.

Wir gehen für die Formeln davon aus, daß das Bild als Helligkeitsmatrix `H[x,y]` abgespeichert ist. Der Wert jedes Pixels liegt im Bereich 0.0...1.0. Dann kann ein Bild folgendermaßen ausgegeben werden:

Abb. 20.12 Floyd-Steinberg-Dither

```
PROCEDURE bildaus;
VAR
 x, y : INTEGER;
 intens : INTEGER;
 error : DOUBLE;
 ...

BEGIN
  FOR y:=0 TO yres-1 DO BEGIN
    FOR x:=0 TO xres-1 DO BEGIN
      intens := map(H[x,y]);    (* Auf diskrete Werte wandeln *)
      drawdot(x,y,intens);      (* Intens ist ein Integer *)
      error := H[x,y] - intens/(stufen-1); (* Fehlerterm *)

      (* Nun Fehler verteilen *)
      H[x+1,y]   := H[x+1,y]    + 7 * error / 16;
      H[x-1,y+1] := H[x-1,y+1]  + 3 * error / 16;
      H[x,y+1]   := H[x,y+1]    + 5 * error / 16;
      H[x+1,y+1] := H [x+1,y+1] + 1 * error / 16;
    END;
  END;
END;
```

Die Funktion `map` ist abhängig von der Zahl der Graustufen, die von der Grafikkarte unterstützt werden. Bei zwei Stufen (Schwarz und Weiß) wäre `map(x) = trunc(x+0.5)`. Man erhält daraus einen ganzzahligen Wert. Nun gilt es, den Fehlerterm zu bestimmen. Dazu muß die Abweichung der gewählten Farbe von der tatsächlichen Farbe berechnet werden. Mit

```
error := H[x,y] - intens / (stufen-1)
```

wird dieser Term ermittelt. Dabei steht in `stufen` die Zahl der Graustufen. Durch die Division wird der Wertebereich wieder auf $0 \ldots 1$ gebracht. Bei

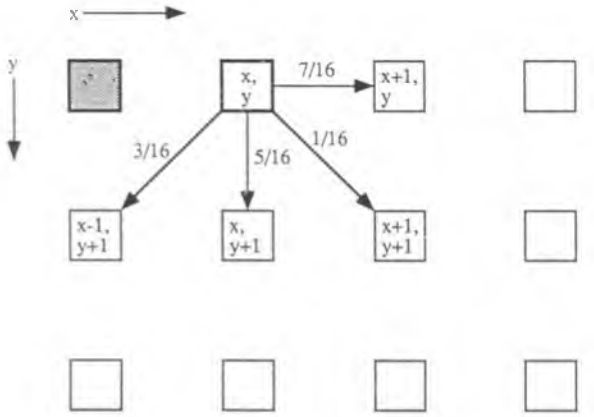


Abb. 20.13 Die Fehlerfortpflanzung beim Floyd-Steinberg-Verfahren

zwei Stufen ergibt sich `error := H[x,y] - intens`, wobei `intens` 0 oder 1 sein kann. Der Fehlerterm wird anschließend auf benachbarte Pixel verteilt. Abb. 20.13 zeigt dazu das Schema.

Dabei werden 7/16 des Fehlers zum rechten Pixel übertragen, 3/16 zum Pixel links unterhalb, 5/16 zum darunterliegenden Pixel und 1/16 zum Pixel rechts unten. Insgesamt also 16/16 = 1. Die Art der Verteilung wurde empirisch gefunden und liefert sehr gute Resultate. Wichtig ist, daß beim Verteilen des Fehlers keine neuen Rundungsfehler entstehen. Gegebenenfalls müssen diese zum letzten Term vorzeichenrichtig addiert werden. Dies ist z.B. nötig, wenn die Helligkeitswerte nicht als Gleitkommazahlen, sondern als Integer-Werte gespeichert sind. Der Bildaufbau erfolgt von links nach rechts und von oben nach unten. Es ist nicht nötig, das gesamte Bild im Speicher zu halten, zwei aufeinanderfolgende Zeilen genügen, um dieses Verfahren zu verwenden.

So wurde es auch im Programm `DITHER2` implementiert. Auch dort wird das Bild zunächst ohne Dithering ausgegeben und dann anschließend mit Dithering überschrieben. Die Routine `zeile0aus` ist nachfolgend abgedruckt und enthält den Algorithmus. Dabei wird genau eine Bildzeile ausgegeben und wie beim „ordered dithering" wird hier mit einer 8-Bit-Grafikkarte gearbeitet, bei der die Farbpalette bereits so initialisiert wurde, daß drei Bits für Rot, drei für Grün und zwei für Blau zur Verfügung stehen (siehe Programm `initpalette`).

```
PROCEDURE zeile0aus(y : INTEGER);
VAR x : INTEGER;
  colorindex : INTEGER;
  r,g,b,r1,g1,b1 : INTEGER;
  errr,errg,errb : DOUBLE;

BEGIN
  FOR x:=0 TO xres-1 DO BEGIN
    r1 := trunc(Hr[x,0] * 7.0+0.5);
    g1 := trunc(Hg[x,0] * 7.0+0.5);
    b1 := trunc(Hb[x,0] * 3.0+0.5);
```

```
      IF (r1 > 7) THEN r1 := 7;
      IF (g1 > 7) THEN g1 := 7;
      IF (b1 > 7) THEN b1 := 7;
      IF (r1 < 0) THEN r1 := 0;
      IF (g1 < 0) THEN g1 := 0;
      IF (b1 < 0) THEN b1 := 0;
      errr := Hr[x,0] - r1 / 7.0;
      errg := Hg[x,0] - g1 / 7.0;
      errb := Hb[x,0] - b1 / 3.0;
      Hr[x+1,0] := Hr[x+1,0] + 7 * errr / 16;
      Hr[x-1,1] := Hr[x-1,1] + 3 * errr / 16;
      Hr[x,1] :=   Hr[x,1]  +  5 * errr / 16;
      Hr[x+1,1] := Hr[x+1,1] + errr / 16;
      Hg[x+1,0] := Hg[x+1,0] + 7 * errg / 16;
      Hg[x-1,1] := Hg[x-1,1] + 3 * errg / 16;
      Hg[x,1] :=   Hg[x,1]   + 5 * errg / 16;
      Hg[x+1,1] := Hg[x+1,1] + errg / 16;
      Hb[x+1,0] := Hb[x+1,0] + 7 * errb / 16;
      Hb[x-1,1] := Hb[x-1,1] + 3 * errb / 16;
      Hb[x,1] :=   Hb[x,1]  +  5 * errb / 16;
      Hb[x+1,1] := Hb[x+1,1] + errb / 16;
      colorindex := r1 + (g1 SHL 3) + (b1 SHL 6);
      moveto(x,y);
      setcolor(colorindex);
      lineto(x,y);
    END;
  END;
```

Das Hauptprogramm sieht dann etwa so aus:

```
...
 hr, hg, hb : ARRAY[-1..320,0..1] OF DOUBLE;
...
 FOR x:=-1 TO xres DO BEGIN (* Init der Matrix *)
   hr[x,0] := 0; hg[x,0] := 0; hb[x,0] := 0;
   hr[x,1] := 0; hg[x,1] := 0; hb[x,1] := 0;
 END;
 FOR y:=0 TO yres-1 DO BEGIN
   FOR x:=0 TO xres-1 DO BEGIN
     r := x / (xres-1);
     g := y / (yres-1);
     b := 1 - r/2 - g/2;
     hr[x,0] := hr[x,0] + r;
     hg[x,0] := hg[x,0] + g;
     hb[x,0] := hb[x,0] + b;
   END;

  zeile0aus(y);
  FOR x:=-1 TO xres DO BEGIN (* Matrix mit Fehlerterm kopieren *)
```

```
      hr[x,0] := hr[x,1];
      hg[x,0] := hg[x,1];
      hb[x,0] := hb[x,1];
      hr[x,1] := 0;
      hg[x,1] := 0;
      hb[x,1] := 0;
    END;
  END;
  ...
```

Das Bild wird Zeile für Zeile aufgebaut und ausgegeben. Nach der Ausgabe jeder Zeile wird die im Feld H befindliche nächste Zeile auf die aktuelle Zeile kopiert und neu mit dem Wert 0 belegt. Bei der Belegung der Zeile wird jeweils der Helligkeitswert aufaddiert, so daß der gespeicherte Fehlerterm erhalten bleibt. Dies geschieht für jede Farbkomponente getrennt und unabhängig.

Die bei diesem Verfahren entstehenden, zufällig verteilten Punkte können ein Bild natürlich auch stören, wenn es sich beispielsweise um technische Zeichnungen mit geometrischen Formen handelt. Dann ist das „ordered dithering" besser. Bei fototypischen Objekten jedoch bietet dieses Verfahren wieder bessere Ergebnisse, da keine störenden geometrischen Artefakte entstehen.

Eine zusätzliche Verbesserung ergibt sich natürlich, wenn eine höhere Bildschirmauflösung verwendet wird.

20.2 PC-Grafikkarten

20.2.1 Super-VGA auf dem PC

Für den PC gibt es die unterschiedlichsten Grafikkarten. Der VGA-Standard ist schon längst durch Super-VGA, XGA etc. erweitert worden. Super-VGA-Karten sind im allgemeinen sehr günstig zu erwerben und haben die Möglichkeit, mit einer höheren Auflösung und mit mehr Farben zu arbeiten als die Standard-VGAs. Leider ist die Art und Weise, wie man auf den Bildspeicher zugreifen kann, von Hersteller zu Hersteller unterschiedlich. Dies kommt daher, daß zur Zeit der VGA-Karten ein Standard für höhere Auflösungen nicht existierte und jeder Hersteller eigene Erweiterungen durchgeführt hat.

Zum Vergleich finden Sie in Tabelle 20.2 die Auflösungen, wie sie bei einer Standard-VGA-Karte vorhanden sind. Die unterschiedlichen Auflösungen lassen sich über ein Modus-Kommando umschalten. Von diesen Auflösungen ist für uns nur der letzte Modus $13 interessant, der eine Auflösung von 320×200 mit 256 Farben erlaubt.

Um die Funktionsweise von Super-VGA-Karten zu verstehen, ist es zunächst am einfachsten, mit diesem Modus zu arbeiten. Der Bildspeicher wird dabei direkt in den Hauptspeicher eingeblendet und kann von Turbo-Pascal über ein absolut adressiertes Feld angesprochen werden. Abb. 20.14 zeigt das Zuordnungsschema.

Tabelle 20.2 Auflösungen der Standard-VGA-Karte

Mode	Mode Sedezimal	Auflösung	Anzahl Farben
4	$4	320 × 200	4
5	$5	320 × 300	4
6	$6	640 × 200	2
13	$D	320 × 200	16
14	$E	640 × 200	16
15	$F	640 × 350	1
16	$10	640 × 350	16
17	$11	640 × 480	2
18	$12	640 × 480	16
19	$13	320 × 200	256

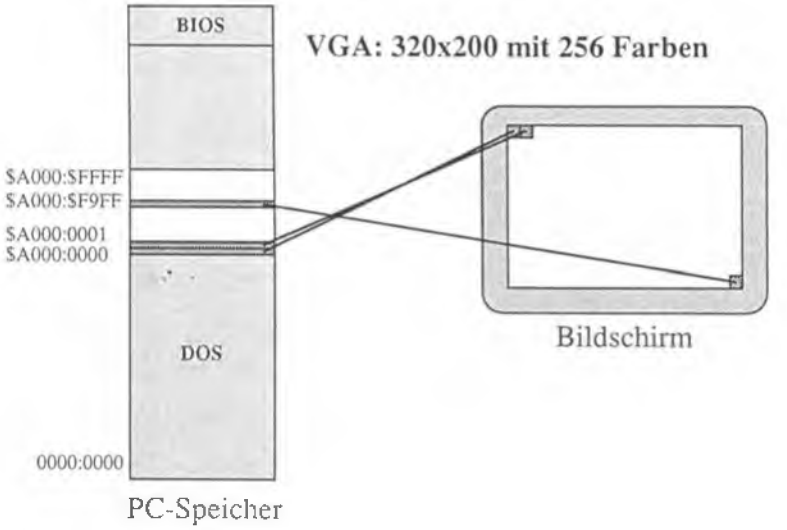


Abb. 20.14 Die Adreßzuordnung VGA und PC

Der Bildspeicher beginnt bei Adresse $A000:0. Pro Pixel wird ein Byte verwendet, also insgesamt 64000 Bytes (320×200). Damit ist die letzte Adresse $A000:$F9FF. Der Bereich $A000:$F9FF bis $A000:$FFFF entspricht einem nicht sichtbaren Bereich des Bildspeichers. Ein kleines Programm zeigt, wie einfach es ist, den Bildspeicher zu adressieren.

```
PROGRAM direkt;
USES crt, dos;
VAR
  x, y   : INTEGER;
  ch     : CHAR;
  color  : BYTE;
  screen : ARRAY[0..199] OF
             ARRAY[0..319] OF BYTE absolute $a000:0;

PROCEDURE setnewmode(mode : WORD); (* Grafik-Mode aktivieren *)
VAR
  i : WORD;
  regs : Registers;
BEGIN
  regs.ah := $0; (* Set Mode *)
  regs.al := mode;
  intr($10,regs);
END;

BEGIN
  setnewmode($13);
  FOR y:=0 TO 199 DO BEGIN
    FOR x:=0 TO 319 DO BEGIN
      screen[y][x] := $2;
    END;
  END;
  FOR y:=80 TO 120 DO BEGIN
    FOR x:=0+31 TO 255+31 DO BEGIN
      color := x AND 255;
      screen[y][x] := color; (* Farbbalken ausgeben *)
    END;
  END;
  ch := readkey;
  setnewmode($2); (* Text Mode *)
END.
```

Mit `setnewmode()` kann ein beliebiger Videomodus gesetzt werden, hier der Modus $13 für 320×200 Punkte. Der Bildspeicher wird als zweidimensionales Feld definiert. Durch die absolute Deklaration des Feldes wird der Speicherbereich direkt auf den physikalischen Bildspeicher abgebildet. Damit ist es möglich, mit `screen[y][x]` von Turbo-Pascal aus direkt auf den Bildspeicher zuzugreifen. Das Hauptprogramm löscht zunächst den Bildspeicher, indem ein konstanter Wert auf alle Speicherpositionen geschrieben wird. Der Wert 2 erscheint dann

Tabelle 20.3 Die Modi des VESA-Standards. Alle Befehle werden über den Interrupt $10 ausgelöst, wobei das Register AX den Wert $4F02 annimmt. Bei erfolgreichem Aufruf wird im Register AL $4F und im Register AH $00 zurückgegeben

VESA 1.1			VESA 1.2		
Modus	Auflösung	Farben	Modus	Auflösung	Farben
$100	640 x 400	16	$10D	320 x 200	32768
$101	640 x 480	256	$10E	320 x 200	65536
$102	800 x 600	16	$10F	320 x 200	16 Mio.
$103	800 x 600	256	$110	640 x 480	32768
$104	1024 x 768	16	$111	640 x 480	65536
$105	1024 x 768	256	$112	640 x 480	16 Mio.
$106	1280 x 1024	16	$113	800 x 600	32768
$107	1280 x 1024	256	$114	800 x 600	65536
$108	80 x 60, Text		$115	800 x 600	16 Mio.
$109	132 x 25, Text		$116	1024 x 768	32768
$10A	132 x 45, Text		$117	1024 x 768	65536
$10B	132 x 50, Text		$118	1024 x 768	16 Mio.
$10C	132 x 60, Text		$119	1280 x 1024	32768
			$11A	1280 x 1024	65536
			$11B	1280 x 1024	16 Mio.

bei normaler Palette als Grün auf dem Bildschirm. Danach wird in einer weiteren Schleife zum Test ein Farbbalken ausgegeben, in dem alle möglichen Farbwerte (0...255) auf 256 x-Positionen geschrieben werden. Der Balken ist 40 Pixel hoch und in der Bildmitte zentriert.

Nach diesem Schema ist es möglich, auf den VGA-Speicher zuzugreifen, ohne die Turbo-Pascal-BGI-Treiber zu verwenden. Das Zeichnen von Linien, Kreisen etc. muß dann allerdings auch selbst programmiert werden. Am Programmende wird mit `setnewmode($2)` wieder der Textmodus des PCs aktiviert.

Super-VGA-Karten haben eine Auflösung, die sehr viel höher liegt als die der Standardkarten. Dazu gibt es neue Modi, die aber je nach Kartenhersteller unterschiedlich sind. Falls Sie also die oben aufgeführten Programme an Ihre Karte anpassen wollen, sollten Sie im Handbuch Ihrer Grafikkarte die entsprechenden Modi und dazugehörigen Auflösungen finden.

Inzwischen gibt es einen neuen Standard für hochauflösende Grafikkarten, der diese unübersichtliche Vielfalt einzudämmen versucht: VESA. Dazu gehören auch neue BIOS-Aufrufe, insbesondere zum Setzen der höheren Auflösungen. In Tabelle 20.3 finden Sie die verschiedenen Modi des VESA-Standards inklusive der Text-Betriebsarten.

Bei allen Super-VGA-Karten wird bei den hohen Auflösungen ein Speicher benötigt, der mehr als 64K umfaßt. Daher kann der Speicher wegen einer Besonderheit des PCs nicht mehr in einem Stück eingeblendet werden. Bei einer Auflösung von 1024 × 768 Pixel sind zum Beispiel bei 256 Farben 786432 Bytes Bildwiederholspeicher vorhanden. Dieser Speicher wird nun einfach in 64K große

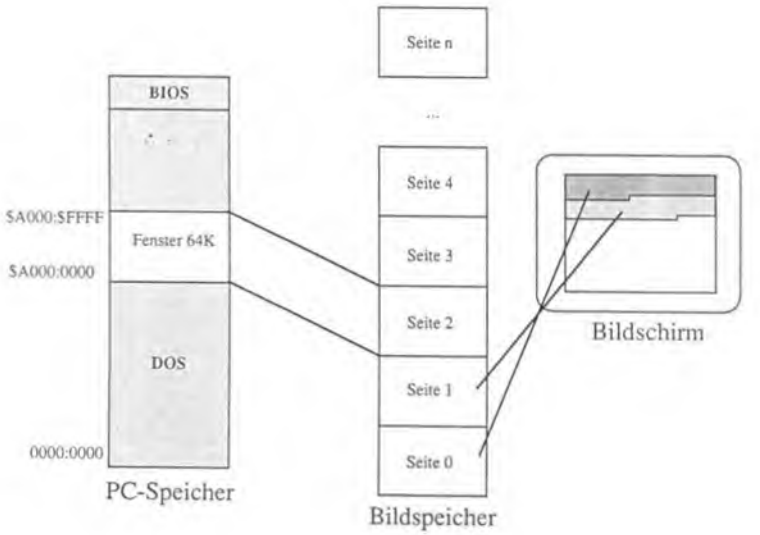


Abb. 20.15 Die Unterteilung des Bildschirms in Seiten

Segmente unterteilt, also hier 12 Segmente, die man auch Seiten (oder Bank) nennt.

Durch Zugriff auf reservierte Register ist es möglich, eine Seite zum Schreiben und dieselbe oder eine andere Seite zum Lesen auszuwählen. Dann findet sich im Bereich $A000:0 bis $A000:$FFFF das angewählte Segment. Abb. 20.15 zeigt das Schema der Umsetzung.

Der Nachteil bei diesem Verfahren ist, daß man nun bei jedem Pixel, das man setzen will, prüfen muß, zu welchem Segment es gehört. Danach müssen die Register gesetzt werden und erst dann darf man auf das Pixel zugreifen. Ein allgemeines Programm für einen solchen Zugriff lautet:

```
totaloffset := x + y * xres
seite := (totaloffset SHR 16) (* DIV 65536 *)
Setze(seite) (* Je nach Karte unterschiedlich *)
speicheradresse := totaloffset AND $FFFF (* MOD 65536 *)
```

Die Einteilung in 64K-Segmente ist leider auch nicht einheitlich. So bieten manche Karten auch die Möglichkeit, mit 128K-Segmenten zu arbeiten oder sie haben andere Startadressen. Die Seite muß natürlich nur dann neu gesetzt werden, wenn ein anderes Fenster eingeblendet werden soll.

20.2.1.1 Bank Select bei ET4000

Der Tseng ET4000-Chipssatz ist sehr verbreitet und daher soll an dieser Stelle der Bankzugriff mit einem praktischen Beispiel vorgestellt werden. Die Zuordnung der verfügbaren Auflösungen bei 256 Farben geschieht anhand der folgenden Moduseinstellungen:

Modus Sedezimal	Auflösung	Anzahl Farben
$2D	640 × 350	256
$2E	640 × 480	256
$2F	640 × 400	256
$30	800 × 600	256
$38	1024 × 768	256

Bei diesem Chip geschieht die Umschaltung über ein einziges Register mit der Adresse $3CD. Dabei steht in den niederwertigen vier Bits die Seitennummer für die Schreibseite und in den höherwertigen vier Bits die Seitennummer für die Leseseite. Damit ist es möglich, von einer Seite zu Lesen und auf eine andere Seite zu Schreiben, ohne das Register zwischendurch umschalten zu müssen. Für den normalen Gebrauch wird allerdings meist Schreibseite = Leseseite gewählt und daher kann man die Funktion Seite() wie folgt angeben:

```
PORT[$3CD] := (seitennr SHL 4) + seitennr
```

Dazu ein Programmbeispiel mit einer Unterroutine fürs Pixelsetzen.

```
PROGRAM pxet4000;
USES crt, dos;
TYPE
  ptrtype = ^byte;
VAR
  x, y : INTEGER;
  ch : CHAR;
  color : BYTE;
  globalxres : LONGINT;
  globalyres : LONGINT;

PROCEDURE setnewmode(mode : WORD); (* Grafik-Mode aktivieren *)

   ... wie oben ...

PROCEDURE setetpixel(x, y : INTEGER; col : BYTE);
VAR
  totaladr : LONGINT;
  ramoffset : WORD;
  screen : ptrtype;
  offset : LONGINT;
  seite : INTEGER;
BEGIN
  totaladr := LONGINT(x) + LONGINT(y) * globalxres;
  seite := (totaladr SHR 16) AND $f;
  offset := totaladr AND $ffff;
  port[$3cd] :=  (seite SHL 4) OR seite;
  screen := ptrtype( $a0000000 + offset AND $ffff);
  screen^:=col;
END;
```

```
BEGIN
  writeln('ET 4000 Mode');
  setnewmode($2E);
  globalxres := 640;
  globalyres := 480;
  FOR y:=0 TO globalyres-1 DO BEGIN
    FOR x:=0 TO globalxres-1 DO BEGIN
      setetpixel(x,y,1);          (* Blauer Hintergrund *)
    END;
  END;
  FOR y:=0 TO globalyres-1 DO
    setetpixel(y,y,15);           (* Diagonale in weiss *)
  setetpixel(0,0,2);           (* linke obere Ecke gruen *)
  setetpixel(639,0,3);     (* rechte obere Ecke tuerkis *)
  setetpixel(639,479,3); (* rechte untere Ecke tuerkis *)
  ch := readkey;
  setnewmode($2);                          (* Text Mode *)
END.
```

Das Programm löscht zunächst den Bildschirm mit `setetpixel`, was allerdings sehr langsam ist, da immer die Berechnungsroutine für die Seitenauswahl aufgerufen wird. Danach wird eine Diagonale von $\begin{pmatrix} 0 & 0 \end{pmatrix}$ nach $\begin{pmatrix} 379 & 379 \end{pmatrix}$ in Weiß gezeichnet und anschließend werden an drei Ecken noch Pixel zur Kontrolle gesetzt. Diese Routine verwendet hier eine Auflösung von 640×480 Pixeln, es kann jedoch auch jeder andere 256-Farben-Modus des ET4000 verwendet werden. Dazu ist die richtige Modusnummer zu setzen und `globalxres` und `globalyres` muß mit der Auflösung besetzt werden.

20.2.1.2 Bank Select bei VESA-Karten

Beim VESA-Standard ist auch an die Bankumschaltung gedacht worden, und es gibt dafür direkt einen BIOS-Aufruf. Achtung: Bei den meisten VESA-Karten muß zuvor ein Treiber geladen werden, um die VESA-Erweiterungen wirksam zu machen. Bitte lesen Sie daher zunächst das Handbuch zur Karte, bevor Sie versuchen, die VESA-Aufrufe zu verwenden.

Der Videomodus wird über einen eigenen INT$10-Aufruf gesetzt. Wenn man feststellen will, ob die VESA-Aufrufe verfügbar sind, so kann man den Rückgabewert der Routinen überprüfen (AL-Register). Die VESA-Aufrufe haben die im folgenden beschriebenen Codes. Alle Befehle werden mit dem Interrupt INT$10 ausgelöst.

1. *Super-VGA-Information zurückliefern*

AX=$4F00	Befehlscode
ES:DI	Adresse für den InfoBlock mit maximal 256 Bytes

   Der Block ist wie folgt aufgebaut:

Offset	Länge	Bedeutung
0	4 Bytes	'VESA'
4	2 Bytes	VESA Versionscode
6	4 Bytes	Adresse der kartenspezifischen Daten
10	4 Bytes	reserviert
14	4 Bytes	Adresse der Modustabelle
18	2 Bytes	Anzahl der vorhanden 64K-Blöcke

Die Modustabelle enthält eine Liste der unterstützten Modi, die mit dem Wert $FFFF abgeschlossen ist. Im Bereich $14 bis $7F sind weitere Modi, die vom Hersteller zusätzlich unterstützt werden.
Die Anzahl der 64K-Blöcke gibt an, wieviel Speicher auf der Grafikkarte vorhanden ist, bei 1MByte sind es z.B. 16 Blöcke. Bei Erfolg steht im Register AL der Wert $4F. Damit kann man auch feststellen, ob eine VESA-Erweiterung verfügbar ist.

2. *Super-VGA-Mode Information*

AX=$4F01	Befehlscode
CX	Super-VGA Modeindex
ES:DI	Adresse eines 256-Byte-Buffers

Mit diesem Aufruf lassen sich Informationen über einen spezifischen Videomodus abfragen. Die Information wird in einem 256 Byte großen Buffer abgelegt. Es ist notwendig, mit diesem Aufruf zu erfragen, wie groß die einzelnen Fenster sind (64K, 128K usw.). Ferner erfährt man, ab welcher Segmentadresse der Bildspeicher eingeblendet wird. Als Ergebnis wird in AX der Wert $004F geliefert und der 256-Byte-Buffer ist mit den folgenden Daten gefüllt.

Offset	Länge	Beschreibung
0	2	Modusattribute
2	1	Attribute von Fenster A
3	1	Attribute von Fenster B
4	2	Fenstergranularität
6	2	Fenstergröße
8	2	Segmentadresse von Fenster A
10	2	Segmentadresse von Fenster B
12	4	Adresse der Fensterroutine für Direktaufruf
16	2	Bytes pro Zeile

Die nachfolgende Information gilt nur dann, wenn Bit 1 der Modusattribute gesetzt ist.

Offset	Länge	Beschreibung
18	2	Horizontale Auflösung
20	2	Vertikale Auflösung
22	1	Breite einer Zeichenzelle
23	1	Höhe einer Zeichenzelle
24	1	Anzahl der Bildebenen
25	1	Anzahl der Bits pro Pixel
26	1	Anzahl der Speicherbänke
27	1	Speicherorganisation
28	1	Größe der Speicherbank in KByte
29	1	Anzahl der verfügbaren Bildseiten
30	1	reserviert
31	1	Maskengröße Rotanteil
32	1	Feldposition Rotanteil
33	1	Maskengröße Grünanteil
34	1	Feldposition Grünanteil
35	1	Maskengröße Blauanteil
36	1	Feldposition Blauanteil
37	1	reservierte Maskengröße
38	1	reservierte Feldposition
39	1	Information True-Color
40	1	reserviert

Das Modusattribut setzt sich wie folgt zusammen:

Bit 0 = 1, wenn der Modus unterstützt wird
Bit 1 = 1, wenn die Zusatzinformationen ab Offset 18 verfügbar sind
Bit 2 = 1, wenn der Modus auch im Standard-BIOS verfügbar ist
Bit 3 = 1, wenn es sich um einen Farbmodus handelt
Bit 4 = 1, wenn es sich um einen Grafik-Modus handelt

Die Bedeutung der Fensterattribute A bzw. B ist:

Bit 0 = 1, wenn dieses Fenster verfügbar ist
Bit 1 = 1, wenn das Fenster lesbar ist
Bit 2 = 1, wenn das Fenster beschreibbar ist

Wenn die Attribute des Fensters A=7 und des Fensters B=0 betragen, so ist nur ein einziges Fenster verfügbar. Bei A=7 und B=7 sind zwei Fenster vorhanden. Mit A=5 und B=3 gibt es zwei Fenster, die jedoch auf demselben Bereich arbeiten, eines zum Lesen und eines zum Schreiben. Mit einem weiteren Aufruf ist es möglich, die Fenster A und B unabhängig voneinander einzublenden.
Die Fenstergranularität gibt den kleinstmöglichen Adreßoffset für ein Fenster an. Bei 8 kann z.B. ein Fenster mit einem Offset von 0K, 8K, 16K

usw. eingeblendet werden. Die Fenstergröße ist der zusammenhängende Bereich, mit dem ein Fenster eingeblendet wird, z.B. 64K. Für jedes Fenster wird auch die Segmentadresse angegeben, bei den Super-VGA-Grafik-Modi z.B. $A000. Ein Eintrag enthält die Adresse der Fensterroutine, so daß es möglich ist, diese durch einen direkten Aufruf zu verwenden. Dies ist nötig, da ein Interrupt mit INT$10 sehr viel langsamer ist, als der direkte Aufruf über einen CALL.
Die zusätzliche Information zur Auflösung ist je nach Modus in Zeichen oder bei Grafik in Pixeln angegeben. Die Anzahl der Bits pro Pixel gibt Information über die vorhandene Farbtiefe, z.B. der Wert 4 für 16 Farben oder 8 bei 256 Farben.
Die Information zur Speicherorganisation ist wie folgt kodiert.

0	Textmodus
1	CGA-Grafik
2	Hercules Grafik
3	Ebenen-Technik (z.B. 16 Bit VGA)
4	Gepackte Pixel wie Standard 256-Farben-Modus
5	Non-Chain 4, 256-Farben-Modus
6	True-Color direkt (HiColor, 24 Bit Farbe)
7	YUV (Luminanz, Chrominanz)
8–$F	VESA reserviert
$10–$FF	kartenabhängig

3. *Super-VGA-Modus setzen*

AX=$4F02	Befehlscode Super-VGA-Modus setzen
BX	Super-VGA-Modus

Durch diesen Aufruf wird der neue Bildschirm-Modus aktiviert. Welche Modi verfügbar sind, muß man über den Aufruf $4F00 bzw. $4F01 vorher überprüfen. Wird Bit 15 in BX auf 1 gesetzt, so wird der Bildschirminhalt nicht gelöscht, sondern der alte Inhalt verwendet.
Bei erfolgreichem Aufruf steht in AX der Wert $004F.

4. *Rückgabe des aktuellen Super-VGA-Modus*

AX=$4F03	Befehlscode

Nach dem Aufruf steht im Register AX der Wert $004F und in BX der aktuelle Videomodus.

5. *Sichern und Retten von Super-VGA-Videozuständen*

AX=$4F04	Befehlscode
DL	Funktionscode
CX	Code für die zu rettenden Bereiche: Bit 0: Hardware Register Bit 1: BIOS RAM Daten Bit 2: DAC Register Bit 3: Super-VGA-Zustand
ES:BX	Adresse des Bufferbereichs (nur bei Code 1 und 2)

Es sind folgende Funktionen möglich.

DL=0	Abfrage der Buffergröße in BX steht die Anzahl der 64-KByte-Blöcke
DL=1	Retten des aktuellen Videozustandes
DL=2	Zurückholen des Videozustands

6. *Seitenkontrolle*

AX=$4F05 Befehlscode

Da nur ein Teil des Bildspeichers im PC-Adreßraum eingeblendet werden kann, muß mit dieser Funktion das Fenster bestimmt werden, das gerade eingeblendet wird. Dabei können zwei Fenster verwendet werden, die mit A und B bezeichnet sind. Je nach Fähigkeit des Grafik-Adapters können diese beiden Fenster z.B. zum getrennten Schreiben und Lesen verwendet werden oder völlig unabhängig sein (siehe Videomodus-Information $4F01).
Es werden zwei Funktionen unterschieden:

(a) *Fenster setzen*

BH	= 0
BL	= 0 für Fenster A = 1 für Fenster B
DX	Offset in Granularitätseinheiten

Danach steht in AX $004F.

(b) *Fenster lesen*

BH	= 1
BL	= 0 für Fenster A = 1 Für Fenster B

Danach steht in AX $004F und in DX der aktuelle Offset, wieder in Granularitätseinheiten.

Bei 640 × 480 mit 256 Farben beträgt die Granularität meist 64K, also wird mit einem Offset von 2 der Bildspeicher ab Adresse 128K angesprochen, was der Pixelposition $x = 512, y = 204$ entspricht.

Das Fenster wird an der Stelle im PC-Hauptspeicher erscheinen, die durch die jeweilige Segmentadresse für Fenster A bzw. B vorgegeben ist (siehe $4F01).

7. *Zeilenlänge setzen oder lesen*

AX=$4F06 Befehlscode

Bei manchen Super-VGA-Karten ist es möglich, eine Zeilenbreite einzustellen, die größer ist als die sichtbare. Damit läßt sich beispielsweise ein horizontales „smooth scrolling" realisieren. Eine andere Möglichkeit ist es, die Zeilenbreite so zu stellen, daß 64K-Segmente niemals innerhalb einer Zeile gewechselt werden, was die Anzeigesoftware vereinfacht. Auch hier gibt es wieder zwei Unterfunktionen.

(a) *Zeilenbreite setzen*
Vor dem Aufruf gilt:

BL	= 0
CX	neue Zeilenbreite in Pixeln

und danach:

AX	= $004F bei Erfolg
BX	Anzahl der Bytes in einer Zeile
CX	Anzahl der Pixel in einer Zeile
DX	Maximale Anzahl an Zeilen (für virtuelle vertikale Zeilen)

Wenn eine Zeilenbreite in Pixeln angegeben wird, die sich nicht genau in eine entsprechende Bytezahl aufteilen läßt, so rundet die Funktion den Wert nach oben. Also z.B. bei 1 Bit pro Pixel mit 8 Pixel pro Byte wird der Wert 801 auf den nächsten möglichen Wert 808 Pixel pro Zeile gerundet.

(b) *Zeilenbreite abfragen*
Vor dem Aufruf muß BL = 1 gesetzt werden, danach ist:

AX	= $004F bei Erfolg
BX	Anzahl der Bytes in einer Zeile
CX	Anzahl der Pixel in einer Zeile
DX	Maximale Anzahl an Zeilen (für virtuelle vertikale Zeilen)

8. *Bildstart lesen oder setzen*

AX=$4F07 Befehlscode

Damit ist es möglich, den sichtbaren Bildstart zu verschieben. So läßt sich „smooth scrolling" in horizontale oder vertikale Richtung programmieren. Auch hier gibt es wieder zwei Funktionsaufrufe:

(a) *Position setzen*

BL	= 0
BH	= 0 (reserviert, muß 0 sein)
CX	Offset in Pixeln horizontal
DX	Offset in Zeilen vertikal

Bei Erfolg wird AX = $004F. Wenn horizontal gescrollt werden soll, muß über die Funktion $4F06 eine virtuelle Zeilenbreite eingestellt werden, die größer ist als die angezeigte, sonst ergibt sich nur ein Rundlaufeffekt.

(b) *Aktuelle Position abfragen*
Beim Aufruf muß BL = 1 gesetzt sein, danach ist

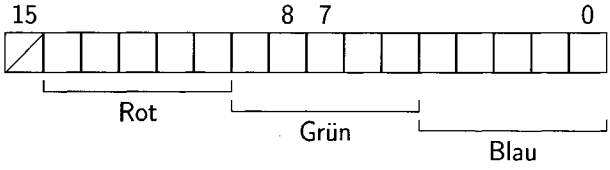
AX	= $004F bei Erfolg
BH	= 0
CX	Offset in Pixeln horizontal
DX	Offset in Zeilen vertikal

Auf der Diskette finden Sie ein komplettes Programm in Pascal, das Informationen über die vorhandene VESA-Karte ausgibt und dann ein Testbild zeichnet (`PXVESA.PAS`). Das Programm gibt zunächst alle verfügbares Modi mit Zusatzinformationen über Auflösung, Anzahl der Farbebenen usw. auf den Bildschirm aus. Dabei wird auf einen Tastendruck gewartet, wenn eine Bildschirmseite fast voll ist. Danach wird der VESA-Mode $101 verwendet, um ein Testbild zu zeichnen.

Die Pixel-Setz-Routine `setvesapixel` kann einzelne Pixel zeichnen und verwendet dazu die VESA-Fensterumschaltfunktion $4F05. Bei der Berechnung der Seitennummer wird hier der Einfachheit halber von 64KByte großen Segmenten ausgegangen. Im Hauptprogramm wird allerdings abgeprüft, ob die Grafikkarte bei dem verwendeten Modus $101 auch 64K-Segmente besitzt. Die Segmentstartadresse wird dynamisch anhand der VESA-Information berechnet.

20.2.2 HiColor-VGA-Karten

Mit einem speziellen Chip können einige Super-VGA-Karten nachgerüstet werden, um dann 32768 verschiedene Farben gleichzeitig darzustellen. Dabei wird pro Pixel mit 16 Bit gearbeitet, wobei für Rot, Grün und Blau jeweils 5 Bit zur Verfügung stehen. Das 16. Bit wird ignoriert.



Bei dem speziellen Chip handelt es sich um den Farbpalettenbaustein, der durch einen DAC 11486 (11481 oder 11488) ausgetauscht wird. Das BIOS der

Super-VGA wie auch die Hardware muß diesen Baustein allerdings unterstützen. Durch eine spezielle Sequenz beim Zugriff auf die internen Register wird die Umschaltung in den sogenannten True-Color-Mode erreicht. Diese Sequenz wird im BIOS durchgeführt und man braucht sich darum nicht weiter zu kümmern.

Es werden dann immer zwei Bytes zusammengefaßt und als Farbwert interpretiert. Der Baustein erlaubt es, mit unterschiedlichen Auflösungen zu arbeiten: $320 \times 300, 640 \times 350, 640 \times 400, 640 \times 480$ und 800×600 Punkten. Falls ein VESA-Treiber mitgeliefert ist, kann man sich über den Videomodus-Informationsaufruf die verfügbaren Auflösungen ausgeben lassen.

Tseng hat ebenfalls BIOS-Erweiterungen definiert, über die sich der HiColor-Modus einstellen läßt. Sie werden bereits von einigen Kartenherstellern unterstützt, u.a. Diamond Computer Systems, SpeedStar HiColor VGA, Everex Systems, HC VGA, Focus Information Systems, 2the Max 4000, Cardinal Technologies, VGA732, Orchid Prodesigner IIs und z.B. der MegaEva1024-Karte (Tseng) , ATI Wonder XL und der Sigma VGA Legend II-Karte.

Die neuen BIOS-Aufrufe werden nun beschrieben. Auch sie werden durch den Interrupt INT$10 gesteuert.

1. *Videomodus setzen*

AX = $10F0	Befehlscode
BL	HiColor-Videomodus

   Danach enthält AX den Wert 0, wenn der Aufruf erfolgreich war.
   Als Videomodus gibt es die folgenden Werte.

$13	$320 \times 200 \times 32768$
$2D	$640 \times 350 \times 32768$
$2E	$640 \times 480 \times 32768$
$2F	$640 \times 400 \times 32768$
$30	$800 \times 600 \times 32768$

   Beim letzten Modus wird fast der gesamte Speicher von 1 MByte benötigt, genaugenommen 960000 Bytes.
   Über einen weiteren Aufruf läßt sich feststellen, ob der HiColor-Baustein vorhanden ist.

2. *HiColor prüfen*

   AX = $10F1 Befehlscode

   Anschließend steht in BL der Wert 0 bei einem normalen DAC (Digital Analog Converter), der Wert 1 bei einem Sierra 1148x-HiColor-Baustein, 2 bei einem neueren Sierra DAC, sonst ein anderer Baustein. Diese Abfrage sollte man daher immer vor dem Setzen eines HiColor-Modus durchführen.

Will man eine Grauskala darstellen, wird man allerdings enttäuscht sein, denn es gibt jetzt nur noch 32 Schattierungen pro Farbe statt der 64 von VGA-Paletten. Doch die Farbenvielfalt wiegt diesen Nachteil schnell auf. Damit dies

nicht nur graue Theorie bleibt, haben wir ein Beispielprogramm, mit dem man den HiColor-Modus einschalten und Pixel zeichnen kann, beigelegt (`PXHICOL`).

Bei der Pixelroutine muß man nun aber darauf achten, daß ein Pixel zwei Bytes belegt, und dies bei der Adreßrechnung entsprechend berücksichtigen. Die Tseng-Karte verwendet auch wieder 64K-Segmente, weshalb die Seitenberechnung und Portbelegung identisch sind mit der ET 4000 für den Standardmodus. Das Programm läßt sich auch leicht für VESA-Karten umschreiben. Dazu muß man nur die Seitenroutine der VESA-Karte entsprechend anpassen.

Das Programm zeichnet zuerst einen farbigen Hintergrund ähnlich wie bei unseren Dithering-Beispielen. Dann werden in einer zweiten Schleife alle möglichen 32768 Farben in einem Fenster ausgegeben. Dabei wird von links nach rechts jeweils zyklisch der Rotanteil erhöht und von oben nach unten zyklisch der Blauanteil. Die einzelnen Kästchen sind immer 32×32 Pixel groß. Zusätzlich wird von links nach rechts und von oben nach unten der Grünanteil langsam von 0 auf 31 gesetzt. Damit werden alle möglichen Farben gezeichnet. Zum Test wird noch in jede Ecke ein Punkt gesetzt, der links oben Weiß erscheinen muß, rechts oben Rot, rechts unten Grün und links unten Blau.

Will man mehr als 32 Graustufen pro Farbe darstellen, so muß man zu Dithering-Techniken greifen. Dazu könnte man die früher verwendeten Programme für „ordered dither“ oder den Floyd-Steinberg-Dither mit der HiColor-Routine kombinieren.

20.2.3 True-Color-Karten

Noch mehr unterschiedliche Farben auf dem Bildschirm erreicht man mit neueren Grafikkarten, die True-Color unterstützen. Sie verwenden eine Farbpalette, die es ermöglicht, immer drei aufeinanderfolgende Bytes zu einem Pixel zusammenzufassen. Damit können dann $256 \times 256 \times 256 \approx 16$ Millionen Farben dargestellt werden. Jedes Byte wird dazu einer Grundfarbe Rot, Grün oder Blau zugeordnet.

Der Baustein ATT20C491/492 ist zum Beispiel eine solche Farbpalette. Dabei wird in diesem True-Color-Modus die eigentliche Farbpalettenfunktion nicht mehr verwendet. Mit Hilfe dieses Bausteins kann man auch im HiColor-Modus und im normalen VGA-Palettenmodus arbeiten. Als Beispiel für Grafikkarten mit True-Color seien hier die Genoa 7900, MegaEva 1024-S und die Diamond SpeedStar erwähnt. Sie verwenden alle den ET4000, dessen Programmierung uns ja bereits bekannt ist. Auch Karten mit anderen Chipsätzen, wie z.B. die Winner 1000 mit dem S3-Chipsatz können True-Color darstellen. In diesem Abschnitt widmen wir uns jedoch dem ET4000 in Zusammenarbeit mit der True-Color-Palette. Die Aufteilung der Bytes im Bildspeicher ist aus dem folgenden Diagramm ersichtlich:

Offset:	0	1	2	3	4	5	
	Blau	Grün	Rot	Blau	Grün	Rot	...

Es folgen immer die Bytes für Blau, Grün und Rot. Eine Besonderheit stellt die Aufteilung des Bildspeichers dar. Beim ET4000 mit den genannten Karten wird True-Color nur bei einer Auflösung von 640 × 480 Bildpunkten unterstützt. Dazu wird ein Speicher von 1 MByte benötigt. Ein Bildzeile belegt hier immer 2048 Bytes statt der für 640 Pixel benötigten 1920 Bytes. Es werden also 128 Bytes verschenkt. Dies ist in der Berechnungsroutine für die Bildposition zu berücksichtigen und nachfolgend dargestellt.

	← 2048 Bytes →	
	Zeile 1	
$A000:$0000	3 Bytes/Pixel je Rot, Grün, Blau	128 Bytes frei
	Zeile 2	
$A000:$0800	3 Bytes/Pixel je Rot, Grün, Blau	128 Bytes frei
	...	
	bis zum Bankwechsel	

Damit kann die Formel

```
pixeladresse = (2048 * y) + x * 3
```

zur Berechnung der Adresse des Pixels auf der Grafikkarte verwendet werden. Da diese Adresse schnell die Größe von 64 KByte übersteigt, wird zusätzlich eine Bankumschaltung verwendet. Die Bankadresse berechnet sich aber einfach zu

```
bank = pixeladresse SHR 16
```

Die tatsächliche Speicheradresse ist dann:

```
adresse = $A000:(0 + (pixeladresse AND $FFFF))
```

Auf der Buchdiskette befindet sich ein Beispiel zur Berechnung der Adresse für die Genoa 7900 mit dem Namen `TRUECOL.PAS`. Die Umschaltung in den True-Color-Modus geschieht wie beim HiColor-Modus über den Interrupt INT$10 mit `AX = $10F0`. Für die einzelnen Grafikkarten ergeben sich jedoch Unterschiede, z.B.:

640 x 480 x 16 Mio. für Genoa 7900:
`INT$10, AX = $10F0, BX = $FF3E`

640 x 480 x 16 Mio. für Diamond SpeedStar 24:
`INT$10, AX = $10E0, BX = $FF2E`

640 x 480 x 16 Mio. für MegaEva 1024-S:
`INT$10, AX = $10F0, BX = $2EFF`

Die Diamond SpeedStar löscht bei der Aktivierung den Bildschirm nicht; die MegaEva benötigt eine zusätzliche Initialisierungssequenz. Wir haben daher hier nur die Genoa 7900 im Programm unterstützt. Die anderen Karten kann man über den VESA-Standard in den entsprechenden Modus versetzen (siehe VESA-Beschreibung in Tabelle 20.3) bzw. die Bankumschaltung mit einem Aufruf vornehmen (siehe VESA-Beispiel auf der Diskette und Abschnitt 20.2.1.2).

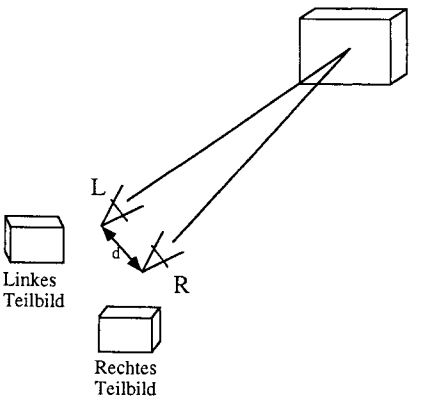


Abb. 20.16 Linker und rechter Bildwinkel für Stereodarstellung

20.3 3D-Darstellung in Stereo

Will man dreidimensionale Bilder auch wirklich plastisch sehen, so muß man zu Stereo-Darstellungsverfahren greifen. Solche Verfahren werden in verschiedenen Anwendungen gebraucht. Zum Beispiel lassen sich damit Molekülmodelle plastisch darstellen. Die Anordnung ist dann viel leichter erkennbar als bei einem flachen Bild. Dies wird zum Beispiel in der Enzymforschung benötigt, um Bindungstellen zwischen einem Enzym und dem dazu passenden Molekül zu finden.

Anwendungen auf dem Gebiet der Medizin ermöglichen die exakte Operationsplanung mit Hilfe von dreidimensional dargestellten Tomografiebildern. Auf dem Gebiet des Maschinenbaus lassen sich mit Stereodarstellungen wesentlich leichter Konstruktionen durchführen als nur mit einem flachen Modell.

Für eine stereoskopische Darstellung ist es nötig, daß linkes und rechtes Auge unterschiedliche Bilder erhalten, und zwar aus der jeweiligen Perspektive des einzelnen Auges. Abb. 20.16 zeigt das Prinzip. Durch den Versatz von linkem und rechtem Auge sieht jedes Auge das dargestellte Objekt unterschiedlich. Die Unterschiede werden vom Gehirn in eine Tiefeninformation umgewandelt.

3D-Stereobilder lassen sich berechnen, indem der Kamerastandpunkt um genau den Augenabstand verändert wird. Man erhält zwei Teilbilder. Jetzt muß man die Teilbilder nur noch so darstellen, daß jedes Auge genau eines der Teilbilder sieht. Das ist leider nicht ganz so einfach, doch es gibt ein paar Verfahren, mit denen man auch mit einem normalen Bildschirm weiterkommt. Zwei dieser Verfahren sollen in diesem Kapitel vorgestellt werden.

20.3.1 Rot-Grün-Brille

Bei diesem Verfahren wird eine Rot-Grün-Brille verwendet, wie man sie z.B. beim Optiker erhalten kann. Die beiden Teilbilder für das linke und rechte

Auge werden dann auf dem Bildschirm entsprechend in Rot oder Grün ausgegeben. Das Auge mit dem grünen Glas sieht nur die grünen Anteile, alle rot gezeichneten Teile sind schwarz und heben sich daher von einem schwarzen Bildhintergrund nicht mehr ab. Umgekehrt werden für das Auge mit dem roten Glas nur die roten Anteile sichtbar und die grünen Bildteile erscheinen schwarz und heben sich daher von einem schwarzen Bildhintergrund nicht mehr ab.

Damit lassen sich nur Schwarzweißbilder stereoskopisch darstellen, da die Bildteile rot bzw. grün eingefärbt werden müssen. Andere Farbanteile dürfen nicht mehr vorkommen, außer den Mischfarben natürlich. Dieses Verfahren läßt sich auf jedem PC mit Farbschirm einfach nutzen und auch die Brillen sind nicht sehr teuer und lassen sich auch selbst herstellen. Leider sieht jedes Auge das Bild entsprechend eingefärbt, was auf Dauer etwas anstrengend ist. Es gibt auch Sehfehler, die verhindern, Bilder nach diesem Verfahren plastisch sehen zu können.

Das PD-Programm `FRACTINT` hat zum Beispiel die Möglichkeit, Fractale dreidimensional auf dem Bildschirm mit Rot-Blau-Technik darzustellen. Ein solches Bild befindet sich auch als Beispiel auf der mitgelieferten Diskette im GIF-Format (`MANDST3D.GIF`). Sie können es mit dem beigefügten Ausgabeprogramm `VPIC` betrachten, wenn Sie eine Rot-Grün-Brille besitzen.

Das Bild ist als Rot-Blau-Bild dargestellt, aber auch mit einer Rot-Grün-Brille betrachtbar. Rot und Blau wird gerne bei Bildschirmen verwendet, da die Trennung durch die Brille für Rot-Blau besser ist als mit Rot-Grün. Die blaue Farbe des Bildschirms hat meist einen nicht unbeträchtlichen Anteil an Grün, aber kaum einen Rotanteil. Das Grün einen Monitors jedoch besitzt auch einen Rotanteil, und dann würden Doppeltbilder entstehen, was sehr viel störender ist.

An dieser Stelle möchten wir Ihnen ein Programm zur Darstellung eines animierten dreidimensionalen Würfels in Stereo vorführen. Der Würfel wird in Rot-Blau-Technik dargestellt. Das Programm finden Sie auf Diskette unter dem Namen `ROTGRUEN.PAS`.

Zur Darstellung des dreidimensionalen Würfels verwendet dieses Programm zwei Bildschirmseiten. Auf einer wird unsichtbar der Würfel in Rot und Blau gezeichnet, während die andere Bildschirmseite dargestellt wird. Nachdem der Würfel fertig gezeichnet ist, wird die Bildschirmseite gewechselt. Dadurch ist eine flickerfreie Animation möglich. Damit sich der Würfel in Echtzeit drehen kann, wird die Rotation mit Hilfe einer Sinus/Cosinus-Tabelle durchgeführt.

Die Prozedur `initsincos` übernimmt die Erstellung dieser Tabellen. Sie wird am Anfang des Hauptprogramms einmal aufgerufen. Die Prozedur `initkoor` setzt einzelne Koordinatenpunkte und wird ebenfalls vom Hauptprogramm aus verwendet, um die Eckpunkte des Würfels zu definieren. Hier können Sie auch leicht Modifikationen zur Darstellung von anderen Gegenständen durchführen. Die Prozedur `drehetrans` führt die Koordinatentransformation und Projektion der Punkte durch.

Jeder Punkt besitzt dazu neben den x-, y- und z-Koordinaten auch die Projektionskoordinaten x_n und y_n. Für jeden Punkt in der Punkteliste wird

zunächst die Rotation durchgeführt.

$$\begin{aligned} x' &= \frac{-z \cdot \text{isin}(\alpha) + x \cdot \text{icos}(\alpha)}{1024} \\ y' &= y \\ z' &= \frac{z \cdot \text{icos}(\alpha) + x \cdot \text{isin}(\alpha)}{1024} \end{aligned}$$

Dabei entspricht $\text{isin}(\alpha)$ der Funktion

$$\text{isin}(\alpha) = \text{trunc}(\sin(\frac{\alpha}{180 \cdot \pi}) \cdot 1024).$$

Die Sinus- und Cosinusfunktion haben dabei einen Wertebereich von -1024 bis $+1024$, die Parameterangabe erfolgt in Grad. Die Abbildung auf den Bildschirm entsteht dann mit der üblichen Projektionsformel.

Die Darstellung des Würfels wird mit Hilfe von Liniengrafik durchgeführt. Die Prozedur `wuerfel` gibt dazu jede Kante des Würfels aus. Im Hauptprogramm wird zunächst der Grafikmodus ausgewählt: Er muß eingegeben werden, denn die hier benötigte Zweiseitendarstellung kann nicht immer durch automatische Anwahl des Grafikmodus definiert sein.

Danach werden die Eckkoordinaten des Würfels definiert und die Sinustabelle programmiert. In der globalen Variablen `w` ist der aktuelle Winkel des Würfels eingestellt. Die Variable `a` wird für die Mehrseitendarstellung benötigt. Zur Bearbeitung von mehreren Bildseiten gibt es in Turbo-Pascal zwei Befehle: `setactivepage(n)` und `setvisualpage(n)`.

Mit `setactivepage(n)` wird die Bildseite ausgewählt, auf die Schreib- und Lesezugriffe erfolgen. Dorthin wird also das Bild gezeichnet. Die Prozedur `setvisualpage(n)` hingegen setzt die Bildseite, die gerade auf dem Bildschirm sichtbar ist. Es ist also möglich, die Zugriffsseite unabhängig von der Betrachtungsseite zu setzen. Das genau wird in unserem Beispielprogramm getan. Mit dem Aufruf `setwritemode(1)` wird zuvor noch der Schreibmodus für alle nachfolgenden Linienbefehle auf X-ODER gesetzt. Dies ist nötig, um beim Zeichnen der blauen und roten Linien auch Mischfarben zu bekommen.

Beim PC ist die Farbpalette so gesetzt, daß beim Überschreiben eines blauen mit einem roten Pixel ein violettes Pixel entsteht. Damit wird dieses Pixel für beide Augen sichtbar. Entsprechendes würde auch für Rot und Grün gelten, jedoch ist beim PC die Mischfarbe nicht das gewünschte Gelb, sondern eher ein Braun mit einem starken Rotanteil. Der Stereoeffekt ist dann nur mäßig. Auch ein Grund, warum hier eine Rot-Blau-Darstellung verwendet wird. Abb. 20.17 zeigt, wie der Bildschirmseitenaufbau erfolgt.

Das Programm zeichnet den Würfel zunächst auf die am Anfang unsichtbare Seite 1. Dabei wird der Würfel einmal in Rot gezeichnet, dann um 10 Grad gedreht und dann in Blau darüber geschrieben. Die Drehung um 10 Grad entspricht etwa dem Versatz der beiden Augen und ist ein kleiner Trick, um sich hier eine aufwendige Transformation zu ersparen. Dadurch, daß die Drehung um den Mittelpunkt des Würfels geschieht, scheint später bei der Darstellung

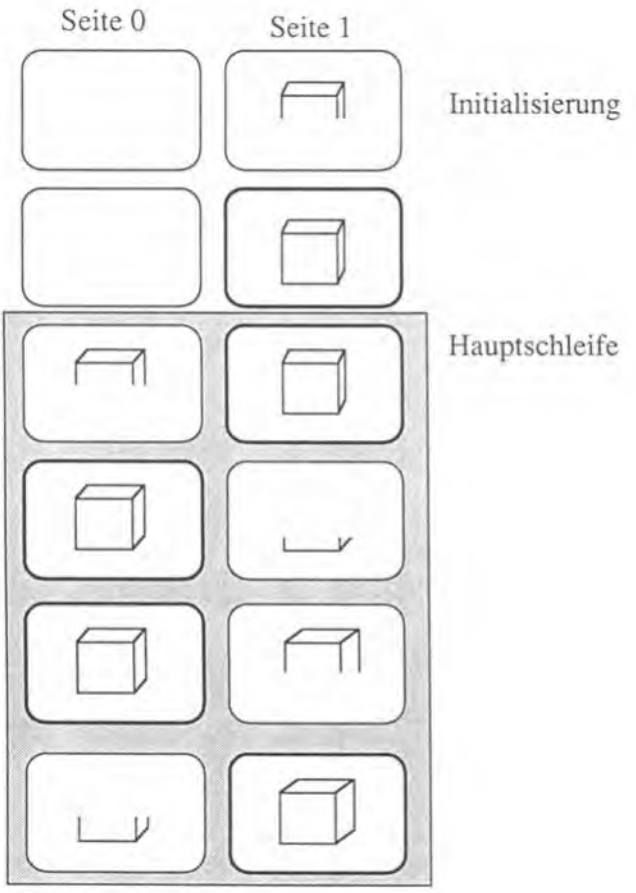


Abb. 20.17 Die Seitenumschaltung beim Beispielprogramm

ein Teil aus dem Bildschirm herauszuragen, während die hinteren Kanten des Würfels hinter der Bildschirmebene liegen.

Nachdem der Würfel dargestellt ist, wird die Bildschirmseite 1 mit `setvisualpage(a)` sichtbar geschaltet. Danach wird der Würfel zunächst in der Schleife wieder um ein Grad gedreht. Mit `setactivepage(a)`, wobei durch die Operation `a = abs(a-1)` der Wert von a wieder auf 0 geschaltet ist, wird Seite 0 als Zeichenseite angewählt. Dort wird zunächst der Bildschirm gelöscht (mit dem `bar`-Befehl, der sehr schnell ist) und dann der Würfel mit der neuen Drehung gezeichnet.

Für den blauen Würfel wird die Position wieder um 10 Grad gegenüber dem roten Würfel gedreht. Dieser ganze Vorgang wiederholt sich, bis die `ESCAPE`-Taste gedrückt wird. Der Würfel dreht sich scheinbar auf dem Schirm. Wenn Sie eine Rot-Grün-Brille besitzen, dann können Sie den Würfel plastisch sehen. Dabei muß man allerdings ein paar Sekunden lang das Bild betrachten, bis sich das Auge an die unübliche Darstellung gewöhnt hat.

20.3.2 LCD-Shutter-Brille

Bei diesem Verfahren wird eine spezielle Brille verwendet, deren Gläser aus Flüssigkristallfeldern bestehen. Mit Hilfe von elektrischer Spannung können die Gläser jeweils durchsichtig oder undurchsichtig werden. Wenn man die Gläser abwechselnd ansteuert, so kann immer nur ein Auge auf den Bildschirm sehen. Auf dem Bildschirm muß man nun dafür sorgen, daß abwechselnd das linke oder das rechte Teilbild sichtbar wird. Synchron dazu werden die elektronischen Gläser gesteuert.

Dieses Verfahren wird auch industriell genutzt, so gibt es z.B. von Crystal Eyes in Kalifornien oder von Tektronix elektronische Shutter-Brillen. Preisgünstige Versionen sind zum Beispiel von Sega oder Haitex für Computerspiele auf den Markt gekommen. Mit diesen Brillen ist es möglich, farbige Stereobilder auf einem Bildschirm darzustellen. Für die Sega- oder Haitex-Brille ist nur eine einfache Schaltung nötig, um die Brillen mit jedem PC zu nutzen. Die Schaltung wird im folgenden zusammen mit Beispielprogrammen vorgestellt.

Sie können damit auch einmal eigene Versuche mit stereoskopischen Darstellungen durchführen. Bei den Low-Cost-Brillen ist leider die Dunkelfärbung nicht so gut, so daß immer noch leichte Teilbilder übrig bleiben. Dennoch ergibt sich ein guter Stereoeffekt, der in jedem Fall besser ist als bei der Rot-Grün-Brille. Der Preis einer solchen Brille liegt dabei zwischen DM 100 und DM 200.

Der Nachteil dieser Brillen im Zusammenhang mit normalen PC-Bildschirmen ist, daß das Bild stark flimmert. Da jedes Auge den Bildschirm nur noch die halbe Zeit sieht, wird aus einem 70-Hz-Bildschirm eine flimmernde 35-Hz-Bildröhre. Im Profibereich werden daher für dieses Verfahren teure Spezialmonitore und Grafikkarten verwendet, die ein Bild mit 100 bis 120 Hz aufbauen können, so daß für das Auge 50 bis 60 Hz übrigbleiben und eine flimmerfreie Darstellung der Stereobilder erlauben.

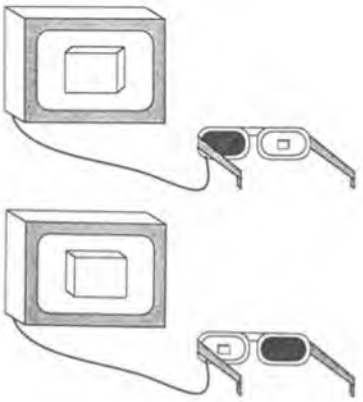


Abb. 20.18 Die Ansteuerung der LCD-Shutter-Brille

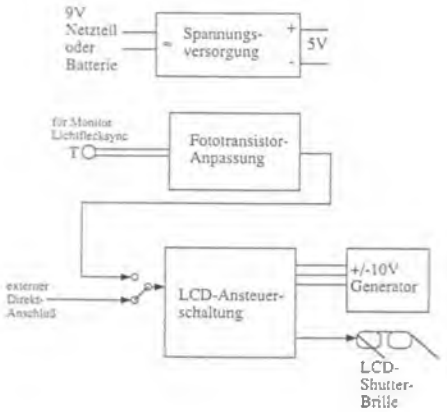


Abb. 20.19 Blockschaltung des LCD-Interfaces

Einige Ausführungen der Bildschirme haben den LCD-Shutter bereits im Bildschirm integriert. Dieser Shutter wird dann bei Aktivierung nicht dunkel, sondern er dreht die Polarisationsebene des Lichts. Der Betrachter muß dann nur noch eine Polarisationsbrille tragen, um das Stereobild zu betrachten.

Abb. 20.18 zeigt das Verfahren mit den aktiven LCD-Shutter-Brillen. In der oberen Bildhälfte ist das linke Auge verdunkelt und nur das rechte Auge sieht den Bildschirm. Dieser zeigt zu diesem Zeitpunkt die rechte Ansicht des 3D-Bildes. In der unteren Bildhälfte ist das rechte Auge verdunkelt, und das linke Teilbild ist sichtbar. Die Umschaltung der Teilbilder erfolgt synchron zum Bildwechsel, so daß keine störenden Flickereffekte auftreten.

Wenn Sie die Sega- oder Haitex-Brille verwenden wollen, so brauchen sie eine Interface-Schaltung. Abb. 20.19 zeigt ein Blockdiagramm dieser Schaltung. Sie enthält auch eine einfache Spannungsversorgung, damit der Anschluß mit einem Steckernetzteil oder einer Batterie möglich ist. Eine Fototransistor-Anpaßschaltung ermöglicht es, die Synchronisation direkt mit dem Bildschirm

durchzuführen. Dazu ist es nötig, per Software einen Synchronisationsfleck auf dem Bildschirm auszugeben. Der Fototransistor wird dann so angebracht, daß er vom Lichtfleck beleuchtet werden kann. Der Lichtfleck wird nur bei dem linken Teilbild ausgegeben.

Diese Schaltung ist hauptsächlich für die Übertragung bei Video- oder Fernsehbildern gedacht, die man dann auch in Stereo betrachten kann. Der schnelle Bildwechsel wird erreicht, indem zwei Bildschirmseiten der Grafikkarte verwendet werden und diese werden dann schnell per Software synchron zum Bildwechsel umgeschaltet. In Turbo-Pascal kann dies z.B. mit dem Befehl `setvisualpage(n)` erfolgen. Auf diese Weise wurden schon 1990 in der Fernsehsendung „Computertreff" Stereobilder ausgestrahlt.

Beim Betrieb mit einem Computer empfiehlt es sich allerdings, möglichst einen direkten Ausgang zur Synchronisation zu verwenden. Dazu gibt es bei der Schaltung auch einen eigenen Eingang. Beim PC läßt sich der Centronics-Druckeranschluß dafür verwenden. Die Softwarebeispiele auf Diskette steuern den Druckerport zusätzlich zur Lichtpunktmethode an. Die LCD-Ansteuerschaltung benötigt $\pm 10V$ zum Betrieb der LCD-Brille. Diese Spannungen werden mit einer integrierten Schaltung aus der 5V-Versorgung erzeugt.

Abb. 20.20 zeigt die komplette Schaltung des LCD-Shutter-Interface. Zum Betrieb der Schaltung müssen zwei Trimmerwiderstände abgeglichen werden. Mit dem Trimmer R7 muß eine Frequenz von ca. 500Hz eingestellt werden. Da Flüssigkristalle nur mit Wechselspannung betrieben werden dürfen, befindet sich auch ein Oszillator in der Schaltung. Der Oszillator besteht aus dem Operationsverstärker U3A. Die Frequenz von 500Hz wird dann an weitere Operationsverstärker geleitet, die daraus ein $\pm 10V$-Wechselspannungssignal erzeugen.

U3B ist dabei für die Seite A und U3C für die Seite B zuständig. Über U2A oder U2C gelangen die 500Hz an die Operationsverstärker. Entweder ist nur am Ausgang von U2A oder nur an U2C die Frequenz vorhanden. So erhält nur ein Brillenglas die Wechselspannung. Die Leitung `trigger` bestimmt, welche Seite dies ist.

Wird die Schaltung direkt vom Computer gesteuert, so bleibt der Schalter SW1 so wie eingezeichnet. Über die Steckverbindung P1 wird dann das Umschaltsignal an die Schaltung geleitet. Abb. 20.21 zeigt den Abgriff von der Centronics-Buchse. Es wird nur die Masseleitung an Pin 25 und das Signal D0 verwendet. D0 wird dazu mit Pin 6 von P1 verbunden und die Masse (0V) mit Pin 8 von P1.

Soll der Fototransistor mit dem Lichtfleck zur Synchronisation verwendet werden, so muß man SW1 umstellen. Dann wird das Umschaltsignal von einem Monoflop U1 erzeugt, das durch den Fototransistor getriggert wird. Das Monoflop ist nötig, da der Lichtfleck nur für kurze Zeit am Bildschirm sichtbar ist und das Umschaltsignal für die gesamte Dauer eines Bilddurchlaufs anliegen muß. Daher muß man das Monoflop auch auf die Bildfrequenz abgleichen. Dazu dient der Trimmer R4. Der Abgleich kann entweder mit einem Oszilloskop oder

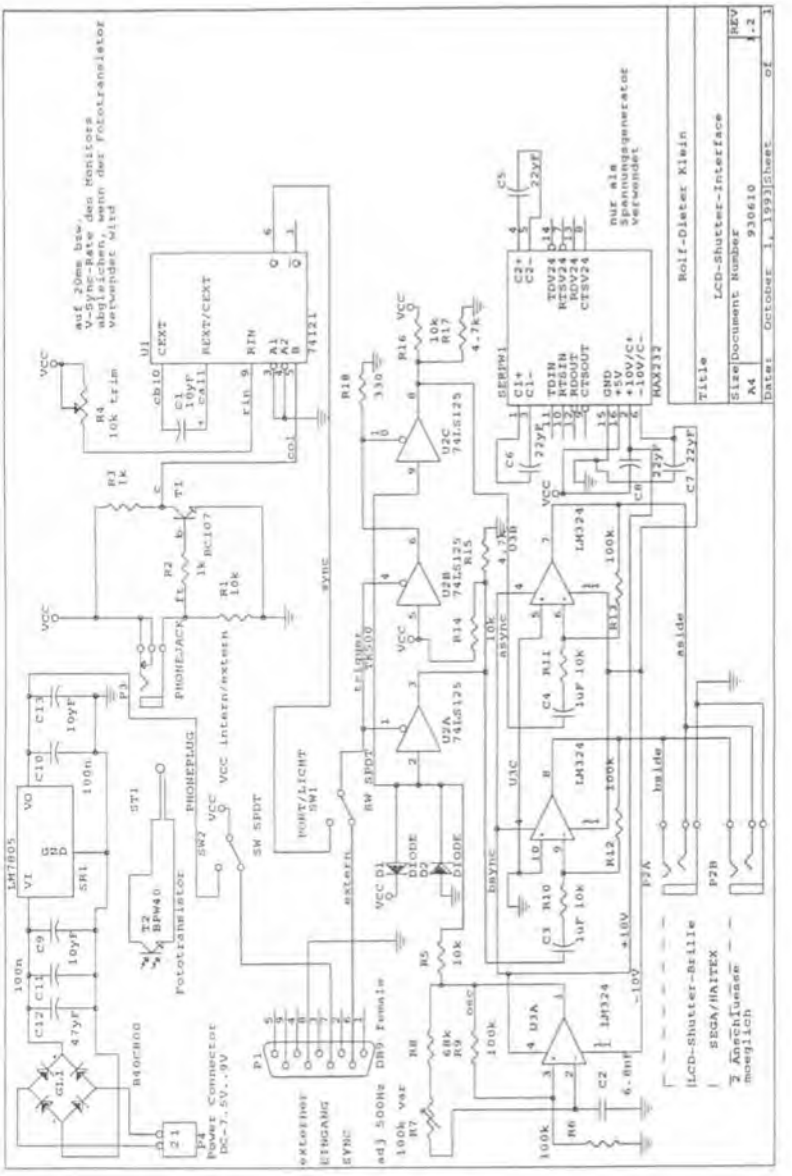


Abb. 20.20 Die Schaltung der LCD-Brille

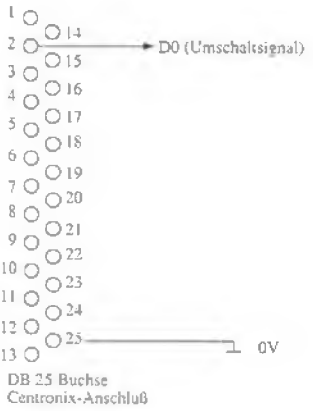


Abb. 20.21 Anschluß über eine Centronics-Buchse

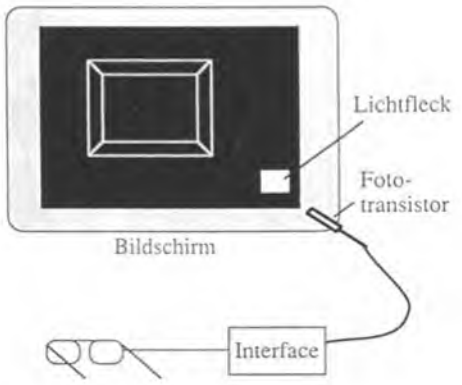


Abb. 20.22 Der Lichtfleck zur Synchronisation

einfach nach Sicht erfolgen, indem das Stereobild betrachtet wird. Man sieht bei falscher Einstellung ein Doppelbild.

Die Zusatzspannung von $\pm 10V$ wird durch den Baustein MAX232 erzeugt, der eine Spannungsvervielfacher-Schaltung enthält. Abb. 20.22 zeigt den Anschluß über den Fototransistor. Der Vorteil liegt darin, daß keine direkte Verbindung zum Computer vorhanden sein muß. So ist es auch leicht möglich, Computer wie den Macintosh oder Workstations zu verwenden.

Die Spannungsversorgung von 5V kann optional durch den Spannungsstabilisator SR1 erzeugt werden, wenn für die Schaltung keine 5V-Versorgung zur Verfügung steht. Dabei genügt ein kleines 9V-Steckernetzteil (ca. 300 mA) oder eine 9V-Batterie.

Doch nun zur Software. Auf der beigefügten Diskette sind einige Programme (`STEREO1.PAS`,...,`STEREO5.PAS`), mit denen sich dreidimensionale Bilder ausgeben lassen. Dabei wird sowohl die Centronics-Schnittstelle angesteuert als auch eine Synchronisation mit einem eingeblendeten Lichtfleck durchgeführt. Das Programm `STEREO1.PAS` erzeugt eine dreidimensionale Linie, die scheinbar

aus dem Bildschirm herausragt. Das Programm ist hier zur Illustration abgedruckt.

```
PROGRAM stereo1;
USES graph, crt;
VAR
  b, a, graphdriver, graphmode: INTEGER;
BEGIN
  graphdriver := detect;
  writeln('1..Grafikmode automatisch waehlen');
  writeln('2..EGALo (640*200,16 Farben)');
  writeln('3..EGAHi (640*350,16 Farben)');
  REPEAT
    write('Bitte waehlen: ');
    readln(a);
  UNTIL a IN [1..3];
  CASE a OF
    2: BEGIN; graphdriver := 3; graphmode := 0; END;
    3: BEGIN; graphdriver := 3; graphmode := 1; END;
  END;
  initgraph(graphdriver,graphmode,'c:\turbo');
  cleardevice;
  setcolor(white);
  bar(getmaxx,getmaxy,getmaxx-50,getmaxy-50);
  setactivepage(1);
  line(310,60,290,200);
  setactivepage(0);
  line(290,60,310,200);
  REPEAT
    WHILE (port[$3da] AND 8)=8 DO;
    REPEAT
    UNTIL (port[$3da] AND 8)=8;
    a:=abs(a-1);
    port[$378]:=a;
    port[$3bc]:=a;
    setvisualpage(a);
  UNTIL keypressed;
  closegraph;
END.
```

Nach Einstellung des Grafikmodus wird zunächst auf Seite 0 mit dem `bar`-Befehl der Synchronisationsfleck rechts unten auf den Bildschirm gezeichnet. Anschließend wird auf die Bildseite 1 umgeschaltet. Dort wird eine Linie gezeichnet. Danach wird wieder die Seite 0 angewählt und eine weitere Linie gezeichnet, die im Raum gedreht ist. Dann folgt die Hauptschleife.

In dieser Schleife wird zunächst durch Abfrage des Ports $3DA gewartet, bis der Bildstrahl die vertikale Austastlücke erreicht hat. Erst dann darf die Bildseite umgeschaltet werden. Dies geschieht mit `setvisualpage(a)`. Zusätzlich wird auf den Druckerports $378 und $3BC die aktuelle Bildseite ausgegeben.

Damit ist die Seitennummer an D0 auf dem Centronics-Stecker erreichbar. Hier werden der Einfachheit halber zwei mögliche Adressen verwendet. Man kann die Ausgabe jedoch auf denjenigen Port beschränken, an dem sich tatsächlich der Centronics-Port befindet. Bei manchen PCs liegt die Schnittstelle auch auf $278, das Programm ist dann entsprechend zu ändern. Der Befehl `a = abs(a-1)` sorgt dafür, daß bei jedem Schleifendurchlauf die Bildseite von 0 nach 1 und umgekehrt wechselt.

Das Programm `STEREO2` arbeitet nach dem gleichen Prinzip, jedoch werden bunte ausgefüllte Kreise auf dem Bildschirm ausgegeben. Diese erscheinen dann für den Betrachter an unterschiedlichen Positionen vor oder hinter dem Bildschirm. Wenn die `RETURN`-Taste gedrückt wird, erzeugt das Programm per Zufallsfunktion eine neue Konfiguration. Mit `ESCAPE` kann das Programm abgebrochen werden.

Das Programm `STEREO3` zeichnet zwei im Raum liegende, aus Kreisflächen zusammengesetzte Schläuche. Bei `RETURN` werden diese in neuer Farbgebung gezeichnet. Das Programm `STEREO4` zeichnet auch „Schläuche", jedoch sind diese zusätzlich im Raum gewunden.

Das Programm `STEREO5` schließlich zeigt einen dreidimensionalen Würfel ähnlich wie im Abschnitt der Rot-Grün-Brille. Bei jedem `RETURN` wird der Würfel in einer neuen Position gezeichnet. Hier ist eine flickerfreie Animation leider nicht möglich, da die beiden zur Verfügung stehenden Bildschirmseiten schon für die Stereoumschaltung verwendet wurden und daher nicht mehr für einen unsichtbaren Bildaufbau zur Verfügung stehen. Bei Grafikkarten mit 4 Bildschirmseiten könnte das Programm jedoch entsprechend verbessert werden.

20.4 Grafik-Beschleunigerkarten

Bisher mußte die Zeichenarbeit immer direkt die CPU des PCs übernehmen. Pixeladressen müssen berechnet werden, eine Bankumschaltung muß vorgenommen werden usw. Dies kostet natürlich Zeit und verlangsamt den Vorgang der Darstellung. In diesem Abschnitt werden zwei Konzepte dargestellt, die zu einer beschleunigten Grafikausgabe führen.

Im ersten Abschnitt wird ein Chip der Firma S3 vorgestellt, der ganz speziell zur Beschleunigung von Windows entwickelt wurde. Er kann dabei bestimmte Zeichenoperationen selbständig vornehmen und so die CPU entlasten.

Im zweiten Abschnitt wird eine Lösung mit dem i860-Prozessor von Intel vorgestellt. Die DSM860-Karte ist eine sogenannte Beschleunigerkarte und kann ganz allgemeine Rechenoperationen übernehmen. Der i860, ein RISC-Prozessor mit einer maximalen Leistung von 120 MIPS kann gleichzeitig bis zu 80 Millionen Gleitkommabefehle und 40 Millionen Festkommaoperationen abarbeiten. Er verfügt außerdem über spezielle Grafikbefehle, wie z.B. Befehle für den z-Buffer-Algorithmus, so wie er in diesem Buch schon besprochen wurde [Mar92]

Der i860 beherrscht aber auch das Gouraud-Schattieren und kann dabei bis zu 40000 schattierte Dreiecke pro Sekunden samt Transformation, Beleuch-

tung, Clippen und z-Vergleich darstellen. Die Dreiecke umfassen dabei etwa eine Fläche von 100 Pixel.

20.4.1 Beschleunigerkarten mit S3

Allgemein unter dem Namen S3 bekannt, gibt es eine Reihe von Grafikkarten, die mit S3-Beschleunigerchips aufgebaut sind. Die einzelnen Chips tragen die Bezeichnungen 86C911, 86C924, 86C801, 86C805 und 96C928. Die Chips unterscheiden sich in der maximalen Bildschirmauflösung (bis zu 1600×1200) und Farbtiefe (4, 8, 15, 16 und 32 Bit/Pixel). Die Befehle sind im wesentlichen für alle S3-Chips die gleichen, jedoch werden 16 Bit pro Pixel erst ab dem 86C801 und 24 bzw. 32 Bit/Pixel erst ab dem 96C928 unterstützt. Ferner sind verschiedene Busse möglich bis hin zum EISA und Local Bus. Die S3-Chips unterstützen VRAMS (Video-RAMs), die einen besonders schnellen Zugriff ermöglichen.

Es gibt dabei Befehle für schnelle Grafikoperationen aber auch für die Cursorsteuerung.

20.4.1.1 Hardware-Cursor

Interessant ist zum Beispiel die Möglichkeit, einen Hardware-Cursor einzublenden. Will man einen Cursor auf einem Bildschirm darstellen, so muß man normalerweise mehrere Schritte per Software durchführen:

1. Retten des Cursorgebietes
2. Zeichnen des Cursors an diese Stelle
3. Warten auf eine neue Cursorposition
4. Restaurieren des Cursorgebietes
5. Weiter bei 1.

Das Verfahren ist in Abb. 20.23 dargestellt. Wird bei Schritt 3 ein Zeichenbefehl durchgeführt, der das Gebiet des Cursors durchkreuzt, dann muß das Gebiet zuvor restauriert und danach der Cursor wieder eingeblendet werden. Man sieht diesen Vorgang z.B., wenn man den Cursor unter Windows über einen Bereich des Bildschirms legt, in dem gerade gezeichnet wird. Der Cursor fängt dann an zu flickern. Der Hardware-Cursor der S3-Karten macht solche Aktionen unnötig. Es wird die Form und die Position des Cursors programmiert.

In diesem Beispiel sieht man auch eine andere Fähigkeit, die für die Fensterverarbeitung wichtig ist: Das Kopieren von Bildausschnitten. Auch diese Operation, der sogenannte BitBlt, wird von den S3-Chips unterstützt und ist komplett in der Hardware implementiert, doch dazu später mehr. Ein Beispiel für eine Software-Animation ist in dem Beispielprogramm `S3EXAMP.PAS` enthalten.

20.4.1.2 Grafikbefehle

Abb. 20.24 zeigt eine Übersicht der von S3-Chips unterstützten Grafikoperationen. Der Linienbefehl ist für einen normalen Prozessor sehr aufwendig.

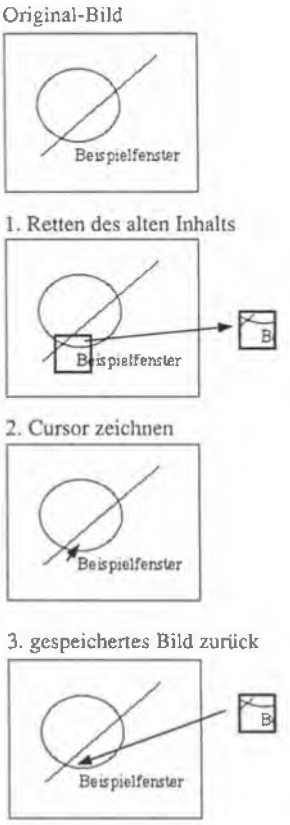


Abb. 20.23 Darstellung eines Cursors

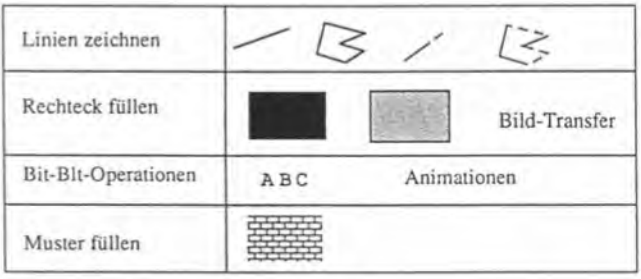


Abb. 20.24 Die Grafikbefehle der S3-Chipfamilie

Zunächst muß die Abfolge der Koordinaten ermittelt werden. Dazu dient z.B. der Bresenham-Algorithmus. Danach muß aus den x- und y-Koordinaten die Speicheradresse und ggf. eine Bitmaske berechnet werden. Außerdem ist es nötig, einen Überlauf der Speicherbank zu überwachen und eventuell eine Bankumschaltung einzuleiten.

Der S3-Grafikbeschleuniger hingegen sieht den gesamten Bildspeicher linear in seinem Adreßraum. Der Linienalgorithmus ist in Hardware gegossen und somit sehr schnell. Vom PC aus muß man dem S3 nur die Start- und Endkoordinaten sowie ein paar Hilfsvariablen und die Farbe mitgeben. Die Abarbeitung wird automatisch durchgeführt. Im Grafikchip werden zwei verschiedenen Linienbefehle unterschieden, die sogenannten Kurzvektoren und Standardlinien. Bei den Kurzvektoren kann man Linien mit bis zu 16 Pixeln in 45°-Schritten von der aktuellen Position zeichnen. Dazu wird nur ein Befehl an den Grafikprozessor in ein Register geschrieben. Der allgemeine Linienbefehl erfordert die Programmierung mehrerer Register und sollte daher nur bei längeren Linien verwendet werden. Für den Linienbefehl befindet sich auch ein Beispiel auf der Buchdiskette in dem Programm `S3EXAMP.PAS`. Das Beispiel sollte für alle S3-Chips verwendbar sind, wurde jedoch nur auf einer Grafikkarte (86C928) mit 8 Bit pro Pixel getestet.

Des weiteren enthält dieser sogenannte GUI-Beschleuniger (Grafical User Interface) einen Befehl für gefüllte Rechtecke. Damit können z.B. Rechtecke in einer beliebigen Farbe aufgefüllt werden. Eine Besonderheit des Chips ist es auch, statt einer festen Farbe auch Daten aus einem durch die CPU auszufüllenden Register anzugeben. Die Bilddaten werden einfach als Datenstrom in das S3-Register geschrieben und vom Chip an die richtige Position in dem Rechteckfenster gesetzt. Damit lassen sich zum Beispiel Fenster aufbauen. Eine Rechteckfläche wird in den Speicher des PCs gerettet und dorthin wird das neue Fenster gezeichnet, ähnlich wie bei dem Beispiel mit dem Cursor. Wird das Fenster vom Benutzer verschoben, so kann man den alten Inhalt restaurieren und dann die neue Position retten. Dieses Verfahren wird z.B. auch bei den sogenannten Pull-Down-Menus verwendet. Wenn Sie ein solches Menu aufklappen, wird das darunterliegenden Bild zunächst gerettet und das Menu darübergezeichnet. Nach erfolgter Auswahl kann das Bild einfach restauriert werden.

Der BitBlt dient zur Übertragung von Bildflächen von einer zu einer anderen Position.

Wie Bildschirmbereiche gerettet und restauriert werden, ist als Beispiel in dem Programm `S3EXAMP.PAS` enthalten. Dort wird mit Hilfe des BitBlts ein 200×200 Pixel großes Feld gerettet und links oben an der Position $(0, 0)$ abgelegt. Dann wird ein mit einem Zufallsmuster belegtes Bild dorthin kopiert. Anschließend wird das Bild wieder restauriert. Dies geschieht so schnell, daß das Auge die Animation als flickerndes Gebilde über den Bildschirm wandern sieht. Das gerettete Gebiet liegt im sichtbaren Bereich links oben und man sieht immer das gerettete Bild „vorbeiziehen". Will man eine flickerfreie Animation, so muß man zwischen Beschreiben des Bildes und der Restauration eine Pause einlegen, z.B. auf das vertikale Synchronisiersignal V-Sync warten.

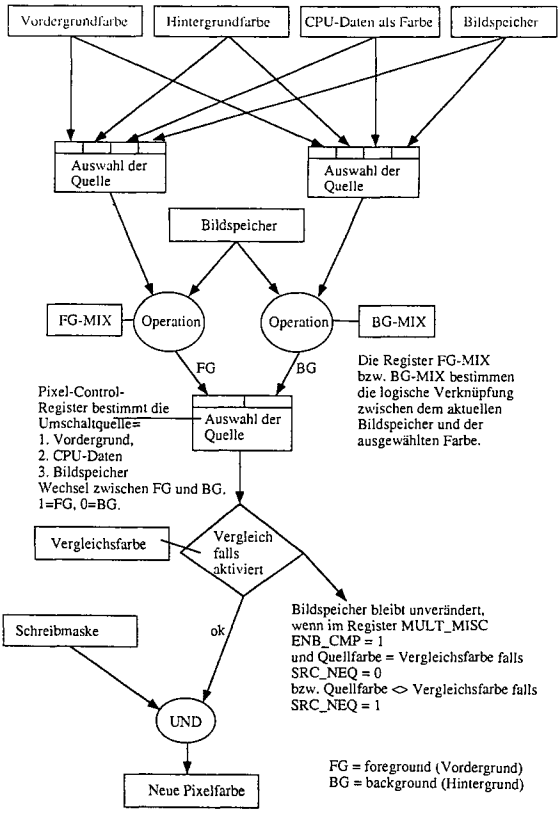


Abb. 20.25 Verknüpfung von verschiedenen Quellen zu einer neuen Pixelfarbe

Der BitBlt muß dann auch schneller als ein Bilddurchlauf sein, sonst sieht man den Restaurationsvorgang. Hier können Sie am Programm einmal selbst weiter experimentieren.

Soll z.B. ein Buchstabe auf den Bildschirm geschrieben werden, so muß der Buchstabe von einer Zeichenmatrix, in der alle Buchstaben gespeichert sind, auf den gewünschten Bildschirmteil verschoben werden. Dies ist zum Beispiel mit einem BitBlt möglich. Auch Fenster und Bildschirmteile für Pull-Down-Menus können mit BitBlts in einen nicht sichtbaren Bereich verschoben werden. Vorausgesetzt es ist genügend Bildspeicher vorhanden, ist das ein sehr schnelles Verfahren.

Der Pattern-(Muster-)Füll-Befehl arbeitet wie der BitBlt-Befehl, nur daß als Quelle ein fester Bereich mit 8×8 Pixeln verwendet wird. Damit können Flächen mit gleichmäßigen Mustern aufgefüllt werden.

Interessant ist es, zusätzlich Verknüpfungen durchzuführen. Es gibt zunächst einmal zwei Farbregister. Eines mit dem Namen Vordergrundfarbe und ein Hintergrundfarbregister. Neben diesen beiden Registern können auch CPU-Daten oder der alte Bildschirminhalt verwendet werden. Abb. 20.25 zeigt ein Schema dazu. Zunächst einmal wird anhand der Clipping-Register entschieden, ob ein

Punkt überhaupt gesetzt werden soll. Dazu sind vier Clipping-Register für die Bereiche TOP, BOTTOM, LEFT und RIGHT eingebaut. Mit einem Bit eines Steuerregisters kann man festlegen, ob nur Punkte innerhalb oder außerhalb des Clipping-Bereiches dargestellt werden. Ist dies der Fall, dann beginnt die eigentliche Berechnung. Im Bild sehen Sie ganz unten die neue Pixelfarbe.

Am besten, wir gehen einmal von unten nach oben vor, um so zu sehen, wie diese Farbe zustande kommt. Ganz am Schluß wird die ermittelte Zwischenfarbe mit der Schreibmaske UND-verknüpft. Nur Bits, die in der Schreibmaske gesetzt sind, werden dargestellt. Davor wird ein Vergleich mit der Vergleichsfarbe durchgeführt. Dieser Vergleich geschieht nur dann, wenn er in einem Register freigeschaltet wird. In diesem Fall erfolgt das Pixelsetzen nur, wenn entweder die aktuelle Bildschirmfarbe mit der Vergleichsfarbe übereinstimmt (`SRC_NEQ = 1`) oder wenn die aktuelle Bildschirmfarbe nicht mit der Vergleichsfarbe übereinstimmt (`SRC_NEC = 0`). Damit lassen sich z.B. Farben auf dem Bildschirm gegen Überschreiben sperren. Wird beispielsweise die Farbe Grün als Vergleichsfarbe eingetragen und `SRC_NEQ = 0` gesetzt, dann werden nur Teile auf dem Bildschirm überschrieben, die nicht grün sind.

Wichtig ist zu beachten, daß die aktuell erzeugte Farbe für den Vergleich keine Rolle spielt.

Es werden grundsätzlich zwei Farbquellen unterschieden, FG (Vordergrund) und BG (Hintergrund). Die Auswahl der Quelle geschieht über ein Steuerregister. Dabei sind drei Möglichkeiten vorhanden. Die erste bedeutet, daß immer die Vordergrundfarbe verwendet wird. Dies ist zum Beispiel bei einfarbigen Linien wichtig. Die CPU, also der PC, schreibt dazu ein Bitmuster in das S3-Datentransferregister. Die einzelnen Bits bestimmen dann, ob die FG- oder die BG-Farbe verwendet wird. Damit lassen sich z.B. strichpunktierte Linien zeichnen. In dem Programm `S3EXAMP.PAS` ist ein Beispiel dazu eingebaut. Die dritte Möglichkeit ist, den Inhalt des Bildspeichers zu verwenden. Dazu wir die Lesemaske genommen und mit den Bits der Farbe des aktuellen Pixels verknüpft. Sind alle Bits auf 1, dann wird die FG-Farbe verwendet, sonst die BG-Farbe.

Die Farben FG und BG werden über eine logische Operation gewonnen. Dazu wird je eine Ausgangsfarbe mit der aktuellen Farbe des Bildspeichers verknüpft. Die Verknüpfung ist für FG und BG getrennt einstellbar. Tabelle 20.4 zeigt die verschiedenen möglichen Operationen.

Für jede der beiden Quellen FG und BG läßt sich außerdem eine Farbe auswählen. Dabei ist es möglich, zwischen vier Quellen zu wählen: Der Vordergrundfarbe bzw. Hintergrundfarbe, die je in einem Register gespeichert sind, den CPU-Daten des Transferregisters und dem Bildspeicher. Wird die CPU als Quelle gewählt, kann man z.B. Bilder vom PC in den Bildspeicher übertragen. Dabei darf die CPU-Quelle nur einmal gewählt werden, also entweder als Farbe oder als Bitanordnung für die Auswahl zwischen FG und BG.

Auf der Buchdiskette ist im Programm `S3EXAMP` auch der Linienbefehl zusammen mit dem XOR-Modus verwendet. Auch ist in dem Beispielprogramm das Zeichnen von Linien mit Strichmustern ein- und zweifarbig eingebaut. Sie können dieses Programm als Basis für eigene S3-Experimente verwenden.

Tabelle 20.4 Die verschiedenen Operationen zur Verknüpfung von Hintergrund- bzw. Vordergrundfarbe (Neufarbe) mit dem Inhalt des Bildspeichers

Operation	Bedeutung
0	NOT Bildspeicher
1	000 ... 0
2	111 ... 1
3	Bildspeicher
4	NOT Neufarbe
5	Bildspeicher XOR Neufarbe
6	NOT Bildspeicher XOR Neufarbe
7	Neufarbe
8	NOT Bildspeicher OR NOT Neufarbe
9	Bildspeicher OR NOT Neufarbe
10	NOT Bildspeicher OR Neufarbe
11	Bildspeicher OR Neufarbe
12	Bildspeicher AND Neufarbe
13	NOT Bildspeicher AND Neufarbe
14	Bildspeicher AND NOT Neufarbe
15	NOT Bildspeicher AND NOT Neufarbe

Tabelle 20.5 zeigt die neuen S3-Register. Zu beachten ist, daß die IO-Ports alle 16 Bit breit sind. Man muß also unter Pascal den `PORTW`-Befehl verwenden. Einige Register sind beim 86C928 auch 32 Bit breit, so zum Beispiel die Vordergrundfarbe, die Hintergrundfarbe, etc. Diese Register verwenden aber nur dann die 32 Bit, wenn auch der Grafikmodus entsprechend auf 24 Bit bzw. 32 Bit pro Pixel eingestellt ist. Dann werden zwei aufeinanderfolgende Schreibzugriffe entsprechend an den niederwertigen und höherwertigen Teil geleitet.

Wer sich näher mit dem S3-Chipsätzen beschäftigt, sollte sich unbedingt das Programmierhandbuch dazu besorgen. Von S3 gibt es auch relativ günstig einen Software-Entwicklungs-Kit (S3 SDK), der die Programmierhandbücher der aktuellen S3-Chips enthält sowie eine C-Bibliothek in Quellform und Programmbeispielen.

Für alle Interessierten daher hier die Anschrift:

S3 SDK Offer
S3 Incorporated
2880, San Tomas Expressway
Santa Clara, CA 95051–0981
U.S.A.

Der S3 SDK kostet außerhalb der Vereinigten Staaten $150. Dazu muß man eine „money order“ an die obenstehende Adresse schicken.

Tabelle 20.5 Die Register der S3-Chipfamilie

Port		Befehl
42E8	R	Subsystem Status
42E8	W	Subsystem Control
4AE8	R/W	Advanced Function Control
82E8	R/W	Current X Position
86E8	R/W	Current Y Position
8AE8	R/W	Destination Y / Axial Step
8EE8	R/W	Destination X / Diagonal Step
92E8	R/W	Error Term
96E8	R/W	Major Axis Pixel Count
9AE8	R	Graphic Processor Status
9AE8	W	Drawing Command
9EE8	W	Short Stroke Vector Transfer
A2E8	R/W	Background Color (16/32)
A6E8	R/W	Foreground Color (16/32)
AAE8	R/W	Write Mask (16/32)
AEE8	R/W	Read Mask (16/32)
B2E8	R/W	Color Compare (16/32)
B6E8	W	Background Mix
BAE8	W	Foreground Mix
BEE8	R	Read Register Data
BEE8/0	W	Minor Axis Pixel Count
BEE8/1	W	Top Clip (Scissors)
BEE8/2	W	Left Clip (Scissors)
BEE8/3	W	Bottom Clip (Scissors)
BEE8/4	W	Right Clip (Scissors)
BEE8/A	W	Pixel Control
BEE8/E	W	Multifunction Control
BEE8/F	W	Read Register Select
E2E8	R/W	Pixel Data Transfer
E2EA	R/W	Pixel Data Transfer Execution

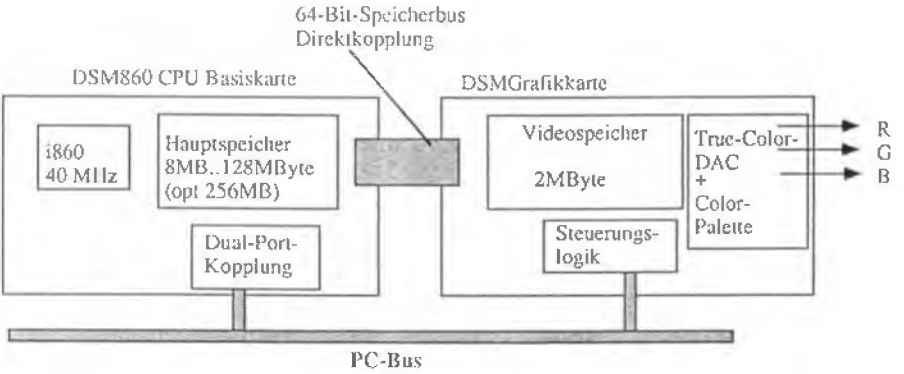


Abb. 20.26 Die DSM860-Karte mit Grafik

20.4.2 Aufbau der DSM860-Beschleunigerkarte

Der Grafikbeschleuniger besteht aus zwei Einsteckkarten, die über Stecker direkt miteinander verbunden sind. Dabei ist die Basiskarte mit der Grafik-Ausgabekarte verbunden. Der i860 kann so direkt auf den Bildwiederholspeicher zugreifen. Ein Pixel hat wahlweise 8 oder 32 Bit Farbtiefe. Der Speicher liegt linear im Adreßraum des i860. Abb. 20.26 zeigt ein Blockdiagramm dazu. Die DSM860 kann je nach Konfiguration mit 8 bis 256 MByte Hauptspeicher ausgerüstet werden. Dort können die Programme und Daten untergebracht werden. Die Grafikkarte besitzt einen Bildwiederholspeicher von 2 MByte. Unterschiedliche Auflösungen lassen sich programmieren. So kann die Karte für True-Color-Darstellung (24 Bit + 8 Bit Overlay) mit 512 × 512 bei zwei Bildschirmseiten und 640×480 oder 768×512 bei einer Bildschirmseite programmiert werden. Im Color-Look-Up-Modus bei 8 Bit pro Pixel sind Auflösungen bis zu 1280 × 1024 bei 110 MHz Punkttakt möglich.

Andere Auflösungen können per Software realisiert werden. Dazu kann die Grafikkarte auch vom PC aus direkt programmiert werden. Die Ansteuerung des i860 erfolgt über Dual-Port-Kopplung. Die Programme auf dem i860 sind bei zeitkritischen Anwendungen in Assembler geschrieben, sonst lassen sich auch C oder Fortran verwenden. Die Anwendersoftware auf der PC-Seite kann unabhängig davon in einer beliebigen Sprache geschrieben sein. Bei Turbo-Pascal erfolgt die Ankopplung zum Beispiel durch die Deklaration:

```
VAR dualmem : ARRAY [0..511] OF BYTE absolute $dc00:0;
```

Danach lassen sich Daten über das Feld `dualmem` mit dem i860 austauschen. Es bildet ein Fenster im PC-Speicher, in das der Speicher des i860 eingeblendet wird. Das Laden des i860-Programms erfolgt mit eigenen Routinen und wird von einem Monitor im EPROM auf der i860-Karte gesteuert.

20.4.2.1 Aufbau des Intel i860 Prozessors

An dieser Stelle soll nur eine kurze Übersicht des i860 und seines Befehlssatzes gegeben werden, damit Sie eine Vorstellung von diesem Prozessor erhalten.

Der i860 gehört zu der Klasse der RISC-Prozessoren, dem aber noch einige Besonderheiten hinzugefügt wurden, wie zum Beispiel der „Superscalar-Mode" (dazu unten noch mehr). RISC steht für „Reduced Instruction Set Computer", also wörtlich: Computer mit verringertem Befehlssatz. Im Gegensatz dazu stehen die CISC-Prozessoren („Complex Instruction Set Computer") für Computer mit umfangreichem Befehlssatz.

Der i860 folgt der RISC-Philosophie, obwohl er auch einige komplexe Befehle besitzt, z.B. für die Grafikverarbeitung. In dieser Architektur wenden sich die Hersteller von der Entwicklung ab, immer komplexere und größere Befehlssätze zu integrieren, von denen nur etwa 20% zur Systemleistung beitragen. Die Reduktion der Befehlssätze und der Befehlsformate trägt erheblich zur Steigerung der Leistung bei. Wichtigstes Merkmal von RISC-Prozessoren ist, daß sie einen Befehl pro Takt ausführen. Die meisten RISC-Prozessoren erreichen dies aber nicht mit Gleitkommazahlen.

Der i860 kann jedoch nicht nur ein Gleitkommaergebnis pro Takt liefern, sondern sogar deren zwei. Und nicht nur das, er bearbeitet zusätzlich im gleichen Taktzyklus noch einen Nicht-Gleitkommabefehl. Diese Fähigkeit nennt man Superscalar-Mode, wie er nur von Superrechnern wie der Cray bekannt ist. Daraus ergibt sich auch die maximale Rechenleistung von 80 Millionen Gleitkommaoperationen pro Sekunde (Million Floating Point Operations per Second, MFLOPS) und 40 Millionen Befehlen pro Sekunde (Million Instructions per Second, MIPS) bei einem Takt von 40 MHz.

20.4.2.2 Die Einheiten des i860

Der Prozessor i860 vereinigt die Fähigkeiten eines Supercomputers mit denen einer 3D-Grafik-Workstation. Er besteht aus 9 Untereinheiten. Abb. 20.27 zeigt das Blockdiagramm des internen i860 Aufbaus.

1. *Die Kerneinheit*
   Die Kerneinheit steuert den gesamten internen Ablauf des i860-Mikroprozessors. Diese Einheit führt alle Lade-, Speicher-, Integer-, Logik-, Bit- und Sprungbefehle aus und stellt die Befehle für die Fließkommaeinheit bereit.
   Für Integer-Daten sind 32 Register mit je 32 Bit Breite vorhanden. Die Lade- und Speicherbefehle können mit 8 Bit, 16 Bit oder auch mit 32 Bit auf diese Register zugreifen und vom Speicher oder zum Speicher transportieren.
   Ein Trap-Mechanismus gestattet Fehlerbehandlung und Reaktion auf einen externen Interrupt. Beim Debuggen gibt es die Möglichkeit, einen Trap auszulösen, wenn auf Instruktionen oder Daten zugegriffen wird.

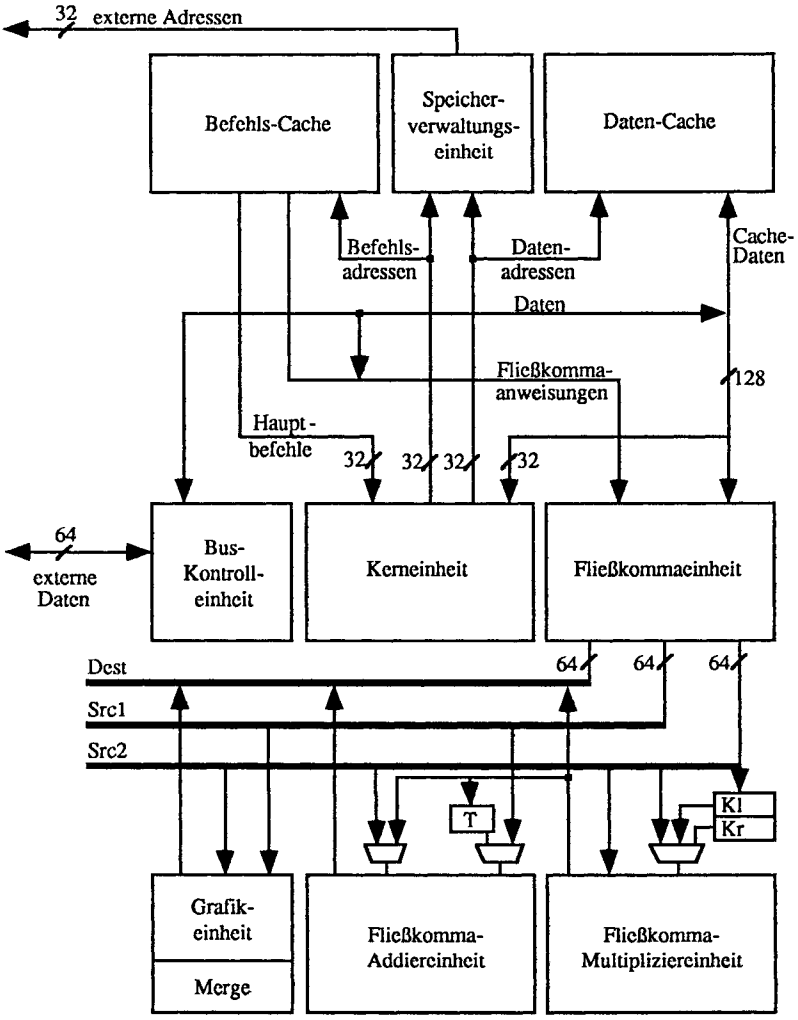


Abb. 20.27 Innenleben des i860

2. *Die Fließkommaeinheit*
   Die Fließkomma-Hardware arbeitet mit einem zweiten Satz von Registern, die als 16 × 64 Bit-Register oder 32 × 32 Bit-Register angesprochen werden. Mit besonderen Lade- und Speicherbefehlen kann man diese Register auch in 8 × 128 Bit-Organisation verwenden.
   Alle Fließkommabefehle verwenden diesen Registersatz. Die Fließkomma-Kontrolleinheit steuert den Fließkommaaddierer und den Fließkommamultiplizierer. Es werden dabei auch alle Ausnahmen (Überlauf, Division durch Null, etc.) behandelt. Den Status der letzten Operation kann man aus den Bits im Fließkomma-Statusregister erkennen. Die Addier- und Multiplizierereinheit können parallel arbeiten und dabei ein oder zwei Ergebnisse pro Taktzyklus produzieren. Die 32 Bit- oder 64 Bit-Fließkommarechnung unterstützt den IEEE-Standard (ANSI/IEEE Std. 754-1985).
3. *Die Fließkomma-Addiereinheit*
   Diese Einheit führt Addition, Subtraktion, Vergleiche und Umwandlungen mit 32-Bit- und 64-Bit-Gleitkommazahlen aus. Die Addiereinheit benötigt drei Takte, um eine Operation komplett auszuführen, jedoch ist es möglich, im Pipeline-Modus ein neues Ergebnis pro Takt zu produzieren. Die drei Operationen werden immer parallel ausgeführt.
4. *Die Fließkomma-Multipliziereinheit*
   Diese Einheit führt Fließkomma- und Integer-Multiplikationen sowie eine Fließkomma-Reziprokberechnung $(1/x)$ mit 32-Bit- oder 64-Bit-Fließkommazahlen durch. Eine Multiplikationsoperation wird in drei bis vier Taktzyklen ausgeführt. Im Pipeline-Modus kann für einfache Genauigkeit (32 Bit Fließkomma) ein Ergebnis pro Taktzyklus berechnet werden. Bei doppelt genauen Fließkommazahlen (64 Bit) steht das Ergebnis nach zwei Taktzyklen zur Verfügung.
5. *Die Grafikeinheit*
   Die Grafikeinheit kann mit Hilfe einer eigenen Integer-Logik dreidimensionale Darstellungen in einem Bildspeicher unterstützen. Dabei werden die verdeckten Kanten mit dem z-Buffer-Algorithmus berechnet. Es ist möglich, eine Farbinterpolation auf Flächen durchzuführen. Die Grafikeinheit kann Pixel mit 8, 16 oder 32 Bit Farbtiefe bearbeiten. Es können die individuellen Rot-, Grün- und Blau-Komponenten für ein Pixel berechnet und mit 64 Bit gleichzeitig bearbeitet werden (also mehrere Pixel pro Zugriff).
   Für den z-Buffer-Algorithmus wird dabei vorausgesetzt, daß die Oberfläche eines Körpers dadurch gezeichnet wird, daß die einzelnen Flächen durch Polygone angenähert werden. Die Lage und Entfernung der Kanten des Polygons werden als bekannt vorausgesetzt, jedoch müssen die Helligkeit und die Abstände zu anderen Punkten interpoliert werden.
6. *Die Speicherverwaltungseinheit*
   Sie ist ähnlich aufgebaut wie die des 80386 und verwendet sogar die gleichen Datenstrukturen. Damit ist es möglich, eine geschützte, sei-

tenorientierte, virtuelle Speicherverwaltung aufzubauen. Logische Adressen können in physikalische Adressen umgesetzt werden. Bei Zugriffsverletzungen wird eine Ausnahmebehandlung gestartet. Es wird auch in Benutzer- und Supervisor-Mode unterschieden.

7. *Der Befehls-Cache*
   Der Befehls-Cache ist 4KByte groß und arbeitet mit 32Byte-Blöcken. Es können pro Taktzyklus bis zu 64 Bit auf einmal übertragen werden – bei 40 MHz Takt sind dies 320 MByte/sec. Ankommende Befehle, deren Speicheradresse nicht im Cache stehen, werden dort zunächst zusammen mit ihrer Adresse zwischengespeichert. Erfolgt ein Zugriff auf eine im Cache vorhandene Adresse, so wird der Inhalt von dort geholt. Der Cache ist dazu als Assoziativ-Speicher ausgeführt, denn der Inhalt des Cache-Speichers muß über seine Adresse gefunden werden. Das ist auch der Grund, warum die Adresse mitgespeichert werden muß.
8. *Der Daten-Cache*
   Der Daten-Cache ist 8KByte groß und ebenfalls als Assoziativ-Speicher ausgeführt. Es können bis zu 128 Bit pro Taktzyklus übertragen werden – bei 40MHz eine Transferrate von 640 MByte/sec. Bei Schreibzugriffen werden die Daten zunächst in den Daten-Cache geschrieben, ohne daß die Werte sofort in den Hauptspeicher übertragen werden. Will man dies vermeiden, kann man das Caching per Software oder Hardware für einzelne Bereiche ausschalten.
9. *Bus-Kontrolleinheit*
   Die Bus- und Cache-Steuereinheit veranlaßt den Daten- und Befehlstransfer für die Kerneinheit. Hier wird der Ablauf für Cache-Treffer, Timing am externen Bus, und die Übersetzung bei virtueller Adressierung durchgeführt. Es können durch die hier vorhandene Pipeline-Struktur bis zu drei Bus-Zyklen anstehen.

20.4.2.3 Register und Befehle

Der i860 besitzt 32 Integer-Register mit jeweils 32 Bit Breite. Das Register R0 hat eine besondere Bedeutung. Wenn es gelesen wird, erhält man immer den Wert 0. Dies spart einige Spezial-Befehle, wie man sie in CISC-Prozessoren benötigt. Ein Transport nach R0 läßt den Wert verschwinden. Damit lassen sich aus Subtraktionsbefehlen elegant Vergleichsbefehle konstruieren, wenn man als Ziel R0 angibt. Darüber hinaus besitzt der i860 32 Gleitkommaregister zu 32 Bit Wortbreite, die sich für doppeltgenaue Zahlen auch als 16 Register mit 64 Bit Wortbreite schalten lassen.

Tabelle 20.6 zeigt eine Übersicht aller i860-Befehle. Wir wollen hier aber nur zwei Gruppen genauer betrachten, die der Dual-Operationen und die Grafikbefehle.

Der i860 kann, wie schon erwähnt, zwei Gleitkommaoperationen in einem Taktzyklus ausführen. Dazu gibt es eine Reihe von Befehlen. Da das Befehlsformat auch nur zwei Quellen und ein Ziel kennt, sind im i860 noch einige Hilfsregister eingerichtet. Mit den Dual-Operation-Mode-Befehlen können nun

Tabelle 20.6 Die Befehle des i860

ld	Integer Laden	fmul	Gleitkomma-Multiplikation
st	Integer Speichern	pfmul	Pipeline-Gleitkomma Mult.
fld	Gleitkomma Laden	pfmul3	3-Stufen Multiplikation
pfld	Pipelined Lade-Befehl	fmlow	Mantissen-Multiplikation
fst	Gleitkomma Speichern	frcp	Gleitkomma Reziprokwert
pst	Pixel ablegen	frsqr	Gleitkomma Wurzel
ixfr	Transport nach Gleitkomma	fadd	Gleitkomma-Addition
fxfr	Transport von Gleitkomma	pfadd	Pipeline-GK-Addition
addu	Addition vorzeichenlos	famov	Gleitkomma-Move
adds	Addition	pfamov	Pipeline-Gleitkomma-Move
subu	Subtraktion vorzeichenlos	fsub	Gleitkomma-Subtraktion
subs	Subtraktion	pfsub	Pipeline-GK-Subtraktion
shl	Links schieben	pfgt	Gleitkommavergleich Größer
shr	Rechts schieben	pfeq	Gleitkommavergleich Gleich
shra	Rechts arithmetisch	pfle	GK-Vergleich Kleiner-Gleich
shrd	Rechts doppeltgenau	fix	Gleitkomma nach Integer
and	Logisches UND	pfix	Pipline GK nach Integer
andh	Logisches UND MSB	ftrunc	Gleitkommastellen abschneiden
andnot	UND NICHT	pftrunc	Pipeline GK abschneiden
andnoth	UND NICHT MSB	pfam	GK Addieren und Multiplizieren
or	ODER	pfsm	GK Subtraktion und Multiplikation
orh	ODER MSB	pfmam	GK Multiplikation und Addieren
xor	Exclusiv ODER	pfmsm	GK Multiplikation und Subtraktion
xorh	Exclusiv ODER MSB	fisub	Long-Integer Subtraktion
trap	Software-Interrupt	pfisub	Pipeline Long-Int Subtraktion
intovr	Überlauf-Interrupt	fiadd	Long-Integer Addition
br	Direkter Sprung	pfiadd	Pipeline Long-Int Addition
bri	Indirekter Sprung	fzchks	16-bit-Z-Buffer Check
bc	Sprung wenn Carry	pfzchks	Pipeline 16-bit-Z-Buffer Check
bc.t	Sprung wenn Carry (taken)	fzchkl	32-bit-Z-Buffer Check
bnc	Sprung wenn nicht Carry	pfzchkl	Pipeline 32-bit-Z-Buffer Check
bnc.t	Sprung w. n. Carry (taken)	faddp	Addieren und Pixel-Mischen
bte	Sprung wenn gleich	pfaddp	Pipeline Addieren und Pixel-Mischen
btne	Sprung wenn Ungleich	faddz	Addieren mit Z-Mischung
bla	Schleifenbefehl	pfaddz	Pipeline Addieren mit Z-Mischung
call	Unterprogramm-Aufruf	form	ODER mit Misch-Register
calli	indirekter Unterprogramm-A.	pform	Pipeline ODER mit Misch-Register
flush	Cache-Speicher leeren	mov	Transport von Registerinhalten
ld.c	Steuerregister laden	fmov	Gleitkomma-Register Transport
st.c	Steuerregister setzen	pfmov	Pipline GK-Register Transport
lock	Bus verriegeln	nop	No Operation
unlock	Bus entriegeln	fnop	Gleitkomma NOP

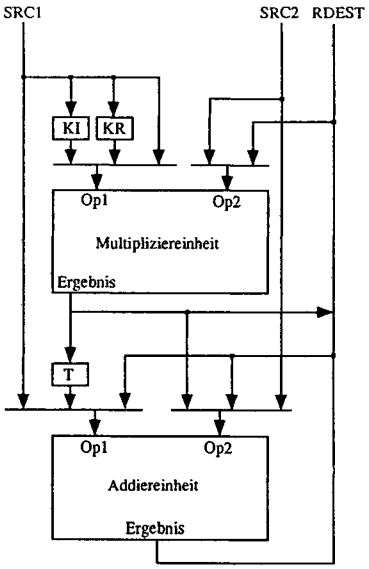


Abb. 20.28 Die Gleitkommaeinheit des i860

auch Daten von diesen Hilfsregistern übernommen oder darin abgelegt werden. KI (Multiplizier-Hilfsregister 1) und KR (Multiplizier-Hilfsregister 2) können den ersten Operanden der Multipliziereinheit ersetzen. T (Addier-Hilfsregister) ist für den ersten Operanden der Addiereinheit bestimmt.

Die Befehle können jeweils ein Ergebnis der Multipliziereinheit oder der Addiereinheit in einem Zielregister ablegen. Das Zwischenergebnis kann auch wieder auf eine Quelle zurückgeführt werden. Es werden vier Klassen von Befehlen unterschieden. Die `PFAM`-Befehlsgruppe (Pipelined Floating Point Add and Multiply) mit der Syntax `xxxx src1,src2,rdest`, die für alle diese Befehle gilt, legt das Ergebnis der Addier-Pipeline im angegebenen Register ab und führt eine Addition im Addierwerk aus.

`PFSM` (Pipelined Floating Point Subtract and Multiply) legt ebenfalls das Ergebnis der Addier-Pipeline im Zielregister ab, führt jedoch eine Subtraktion mit den neuen Operanden im Addierwerk durch. Die Gleitkommaeinheiten des i860 sind in Abb. 20.28 detailliert dargestellt.

`PFMAM` (Pipelined Floating-Point Multiply with Add) legt entsprechend das Ergebnis der Multipliziereinheit im Register ab und führt eine Addition im Addierwerk durch. `PFMSM` (Pipelined Floating Point Multiply with Subtract) legt auch das Ergebnis der Multipliziereinheit im angegebenen Zielregister ab, führt aber eine Subtraktion mit den Operanden im Addierwerk durch. Es muß darauf geachtet werden, daß bei doppelt genauen Gleitkommazahlen (64 Bit) die Multiplizier-Pipeline nur zwei Stufen lang ist, während die Addier-Einheit immer noch eine Länge von drei Stufen hat. Dies ist bei der Verwendung dieser Befehle zu berücksichtigen. Bei einfacher Genauigkeit (32 Bit) sind beide Pipelines drei Stufen lang.

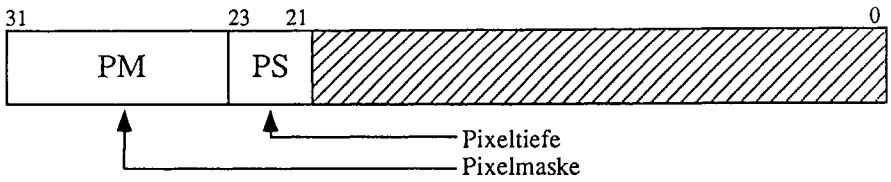


Abb. 20.29 Die Belegung des Steuerregisters für Pixel

20.4.2.4 Die Befehle für die Grafik-Recheneinheit

Die Grafik-Recheneinheit arbeitet im Gleitkomma-Registersatz. In ihm werden 32-Bit- oder 64-Bit-Integer-Operationen durchgeführt.

Will man dreidimensionale Gegenstände auf dem Bildschirm plastisch mit verdeckten Kanten darstellen, kann man verschiedene Verfahren anwenden. Raytracing ist eine Möglichkeit, für die man z.B. beim i860 die Gleitkommabefehle sehr gut einsetzen kann. Der z-Buffer-Algorithmus ist nicht so rechenaufwendig, erzeugt aber dennoch eine hohe Bildqualität. Der z-Buffer-Algorithmus ist im i860 mit eingebaut. Zunächst muß dem i860 jedoch die Art des Bildspeichers mitgeteilt werden. Dies geschieht über ein Prozessorregister.

Im Prozessor-Statusregister wird die Pixeltiefe (8 Bit, 16 Bit oder 32 Bit) festgelegt. Der i860 ist in der Lage, Bildschirmspeicher mit unterschiedlicher Organisation anzusteuern. Bei 8 Bit Tiefe kann man mit 256 Graustufen arbeiten, oder über eine Farbpalette gehen. Die 16- und 32-Bit-Modi sind für Echtfarbendarstellung gedacht. Dazu wird ein Wort entsprechend unterteilt. Im Feld PM (Pixel Mask) wird die Information, welche Punkte verändert werden sollen, abgelegt. Die Maske wird durch die z-Buffer-Befehle direkt verändert. Abb. 20.29 zeigt den Aufbau dieses Registers. Wenn dieses Register definiert wurde, können die i860-Grafikbefehle verwendet werden.

Die i860-Befehle setzen die Pixelmaske in Abhängigkeit vom Vergleich der z-Buffer. Dazu gibt es zwei verschiedene Befehle. Der `fzchks`-Befehl (z buffer check short) vergleicht mit 16 Bit tiefen z-Buffern. Vier Punkte werden auf einmal verglichen. Der Inhalt des alten z-Buffers steht zuvor in `src1`, der zu vergleichende Wert in `src2`. Anschließend wird in `rdest` der kleinere von beiden Werten abgelegt. Die Pixelmaske ist 8 Bit groß. Sie wird vor dem Vergleich um vier Bit nach rechts verschoben. Wenn das entsprechende Feld in `src1` kleiner als `src2` war, so wird das dazugehörige Bit in der Pixelmaske gesetzt.

Bei dem `fzchkl`-Befehl (z buffer check long), dem zweiten Befehl, werden zwei 32 Bit-Werte verglichen. Entsprechend wird die Pixelmaske nur um zwei Bit nach rechts verschoben. Nur zwei Bits werden entsprechend dem Vergleich gesetzt. Mit dem so gewonnenen Ergebnis im Feld PM können nun weitere Befehle der Grafikeinheit verwendet werden, um die Punkte im Bildwiederholspeicher abzulegen. Für die Interpolation der z-Werte stehen eigene Befehle zur Verfügung.

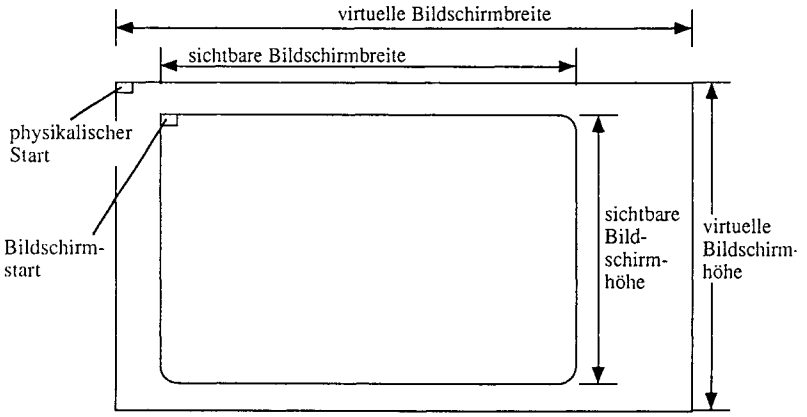


Abb. 20.30 Virtuelle Bildschirme

20.4.2.5 Zugriff auf den i860-Bildspeicher

An dieser Stelle soll ein kurzes Programm zeigen, wie einfach mit einem linearen Bildspeicher Zugriffe auf den Bildspeicher erfolgen können. Da der Speicher nicht wie beim PC in Segmente unterteilt ist, entfällt eine Abfrage der Segmentgrenzen. Es gibt nur noch zwei wichtige Parameter. Die Startadresse des Speichers und die Anzahl der Bytes pro Zeile, um für die y-Koordinate eine Adresse ausrechnen zu können. Ferner muß man noch wissen, ob ein Byte, ein Wort oder ein Langwort pro Pixel verwendet werden soll. Damit reduziert sich die Pixel-Setzroutine auf eine Befehlszeile.

```
(ADRESSE + x + y * BREITE)^  := COLOR;
```

Die Multiplikation kann noch durch einen Schiebebefehl ersetzt werden, wenn die Anzahl der Bytes pro Zeile eine Zweierpotenz ist.

Als Adresse wird beispielsweise $A0000000 und als Breite 2048 eingesetzt. Bei einer Darstellung von 1280 Pixeln pro Zeile im 8 Bit-Modus bleiben die restlichen 768 Punkte unsichtbar. Diese können jedoch durch Extra-Register per „soft scroll" sichtbar gemacht werden. Abb. 20.30 zeigt die Aufteilung bei einem virtuellen Bildschirm. Eine physikalische Startadresse bestimmt den Anfang des Bildspeichers. Das Bild wird jedoch erst ab einer Bildschirm-Startadresse, der logischen Adresse ausgegeben. Auch muß die Bildschirmbreite nicht mit der Breite des physikalischen Bildspeichers übereinstimmen. Man unterscheidet dabei die virtuelle Bildschirmbreite und die logische Breite des Bildschirms. Das gleiche gilt für die Höhe des Bildspeichers. Wenn die Position der logischen Startadresse programmierbar ist, so können vertikales und horizontales Scrolling durchgeführt werden. Man spricht von „soft scroll", wenn die Verschiebung pixelweise durchgeführt werden kann.

Mit einem virtuellen Bildschirm ist der Computer in der Lage, ein Bild zu zeichnen, das größer ist als sein sichtbarer Teil. Durch Ändern der logischen Startadresse des Bildschirms kann dann anschließend vom Benutzer der ganze Bereich durchlaufen werden. Der Vorteil liegt insbesondere bei ganz großen vir-

tuellen Bildspeichern darin, daß man das Bild nur einmal zeichnet und beim Umherwandern nur die Startadresse ändern muß.

VI

Anhang

Programmierung von Rendering und Radiosity

Josef Pöpsel
Ute Claussen

```
⋮
procedure ...
      ⋮
end;
⋮
```

A. Beschreibung der Eingabesprache GED

Um die Definiton von darzustellenden Szenen möglichst einfach zu gestalten, existiert eine Unit, die in der Lage ist, Szenenbeschreibungen, die als ASCII-Dateien vorliegen, in die internen Datenstrukturen der Programme umzusetzen. Der Name dieser Unit ist `PRIMREAD`.

In Anlehnung an „Grafische Eingabe Datei" haben solche Szenenbeschreibungsdateien die Endung „.GED". Sie werden mittels der Routine `LeseGED-Datei` eingelesen. Dieses Prozedur baut aus der Eingabedatei eine Baumstruktur der Szene auf. Das aufrufende Programm gibt die dann erhaltene Struktur an die Routine `DereferenziereSzene` weiter, die aus der Baumstruktur eine Liste von Punkten und eine Liste von Flächen erzeugt. Diese Listen können besonders einfach dargestellt werden. Zusätzlich wird von `LeseGEDDatei` eine Baumstruktur aufgebaut, die verwendet werden sollte, wenn das Strahlverfolgungsprogramm in `PRIM` integriert werden soll. Das Schema dieses Ablaufes ist Abb. A.1 zu entnehmen.

In der Unit `PRIMREAD` ist nur eine einzige Prozedur öffentlich definiert:

```
PROCEDURE LeseGEDDatei(filename            : STRING;
                       VAR SzenePtr        : ObjektPtrTYP;
                       VAR LichtquellenPtr : LichtquellenPtrTYP;
                       VAR Betrachtung     : BetrachterTYP;
                       VAR WC_NPC, NPC_WC,
                           NPC_DC, WC_DC   : Mat4TYP;
                       VAR Hintergrund     : HintergrundTYP);
```

Diese Prozedur wandelt die Szene, die in `filename` beschrieben ist, in Datenstrukturen für Objekte, Lichtquellen, Transformationen und die Beschreibung des Hintergrundes um. Bevor auf diese Datenstrukturen eingegangen wird, folgt eine Definition der Eingabesprache. Die genaue Syntax der Eingabesprache ist in Anhang B beschrieben.

A.1 Die Eingabesprache GED

Will man Szenen beschreiben, so unterteilt sich diese Aufgabe in verschiedene Unteraufgaben:

- Die geometrische Beschreibung der darzustellenden Objekte.

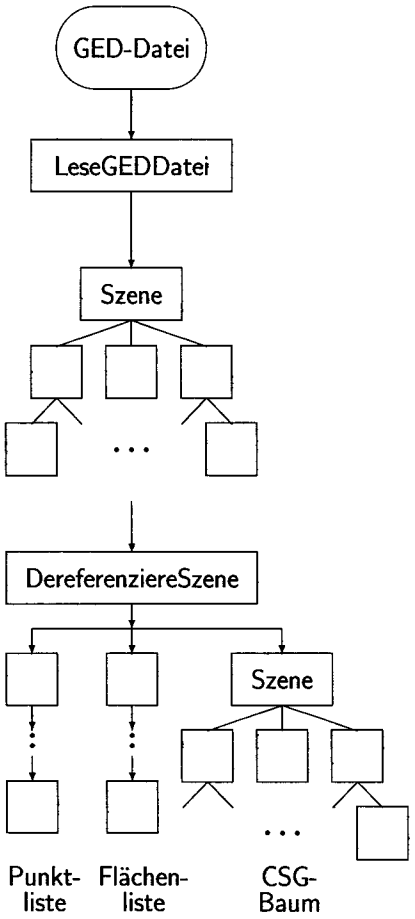


Abb. A.1 Ablauf des Einlesens der GED-Dateien

- Die Beschreibung der Oberflächeneigenschaften dieser Objekte.
- Die Lage der Objekte zueinander.
- Die Beschreibung des Blickpunkts, der Blickrichtung u.ä.
- Die Beschreibung der Lichtquellen, die die Szene beleuchten.
- Die Beschreibung des Hintergrundes einer Szene.

Entsprechend dieser Unteraufgaben existieren in einer GED-Datei Teile, in denen genau die angesprochenen Beschreibungen angegeben werden. Ohne die Syntax hier korrekt wiederzugeben, sieht die Beschreibung einer Szene etwa so aus:

```
GED_VERSION Versionsnummer;

Objektliste
Betrachter
Lichtquellen

SZENE SzenenName;
   Hintergrundbeschreibung
   Teilobjektliste
END.
```

A.2 Objekte

Objekte sind Zusammenfassungen von Teilobjekten, die beim Aufruf eines Objektes gemeinsam dargestellt werden. Teilobjekte werden weiter unten erläutert.

Die Syntax für ein Objekt hat die Form:

```
OBJEKT name;
  Teilobjekt
  Teilobjekt
  .
  .
ENDOBJEKT;
```

Dabei steht **name** für eine beliebige Zeichenkette. Mit Hilfe dieses Namens kann das Objekt z.B. in anderen Objekten als Teilobjekt wieder aufgerufen werden. Damit kann u.a. auch eine Vererbung von Eigenschaften erreicht werden, ein wesentliches Merkmal objektorientierter Sprachen [Cla93].

Ein besonderes Objekt ist die SZENE. Die Szene muß immer am Ende einer GED-Datei definiert sein. Sie wird letztlich dargestellt. Vergleicht man OBJEKT und SZENE mit einem Pascal-Programm, so entspricht OBJEKT einer Prozedur und SZENE dem Hauptprogramm.

Die Syntax für die SZENE weicht nur unwesentlich von der eines Objektes ab:

```
SZENE name;
  [Hintergrundobjekt]
  Teilobjekt
```

```
  Teilobjekt
  .
  .
END.
```

Der name der SZENE wird nicht weiter verwendet.

Eine vollständige GED-Datei besteht mindestens aus einer SZENE. Objekte müssen nur dann definiert sein, wenn sie in der SZENE direkt oder indirekt verwendet werden. Das Hintergrundobjekt dient dazu, im Hintergrund der Szene ein bestimmtes Bild oder einen Farbverlauf darzustellen. Fehlt es, so wird ein vordefinierter Farbverlauf im Hintergrund verwendet. Die Syntax des Hintergrundobjektes wird weiter unten beschrieben.

Außer der Szene ist die Angabe einer Versionsnummer am Anfang einer GED-Datei zwingend. Dies dient dazu, für eventuelle Erweiterungen der Eingabesprache gewappnet zu sein. Die Syntax der Versionsbeschreibung ist:

```
GED_VERSION zahl;
```

Dabei entspricht zahl der aktuellen Version, derzeit 1.

Ein sehr einfaches Beispiel für eine korrekte GED-Datei ist:

```
// Eine erste Testszene

GED_VERSION 1;
GERICHTETELQ 1,1,1 1,1,-1;
SZENE Eine_Ganz_Einfache_Szene;
  KUGEL;                  // Kugel ist ein vordefinierter Koerper
END.
```

Es wird nicht zwischen Groß- und Kleinschreibung unterschieden. Kommentare beginnen in C++-Manier mit „//“. Alles von dort bis zum Zeilenende wird überlesen.

Alle oben aufgeführten Beschreibungen einer Szene (Betrachterstandpunkt, Oberflächeneigenschaften etc.) sind optional und erhalten, wenn sie nicht explizit gesetzt sind, vordefinierte Werte. Eine Zusammenfassung aller vordefinierten Werte einer GED-Datei ist am Ende dieses Anhangs zu finden.

A.3 Teilobjekte

Ein Teilobjekt besteht aus einer geometrischen Beschreibung, der optional Transformationen und Oberflächenbeschreibungen vorangestellt werden können. Der Unterschied zwischen Teilobjekten und Objekten ist, daß

- nicht auf Teilobjekte über eine Referenz zugegriffen werden kann.
- bestimmte Berechnungen nur innerhalb von Teilobjekten, nicht aber teilobjektübergreifend, also objektweise erfolgen. Zu diesen Berechnungen gehört z.B. die Bestimmung von Nachbarschaften bei polygonalen Teilobjekten.

Es existieren vier verschiedene Arten von Teilobjekten: polygonale Teilobjekte, Referenzen auf Objekte, vordefinierte Körper und CSG-Ausdrücke.

A.3.1 Polygonale Teilobjekte

Bei der Polygonbeschreibung handelt es sich um die geometrische Beschreibung durch planare konvexe Vielecke. Als erstes werden in einer Polygonbeschreibung Punkte, anschließend Flächen definiert. Dabei werden in der Flächenliste die vorher beschriebenen Punkte durch ihren Index angesprochen. Der erste Punkt der Punktliste hat den Index Null, der zweite den Index Eins u.s.w.

Ein einfaches Beispiel für eine Polygonbeschreibung ist:

```
PUNKTE
  1,1,0;
  -1,1,0;
  -1,-1,0;
  1,-1,0;
FLAECHEN
  0,1,2,3;    // Definition des Vierecks
```

Das Beispiel besteht also nur aus einem Viereck.

A.3.2 Referenzen auf Objekte

Die Referenz auf ein Objekt entspricht in etwa einem Prozeduraufruf in Pascal. Die Syntax hat die Form:

```
ZEICHNE name;
```

wobei `name` ein Objekt ist, das vor diesem Teilobjekt definiert wurde.

A.3.3 Vordefinierte Körper

Vordefinierte Körper sind Objekte, die im Normalfall nicht aus Polygonen zusammengesetzt sind. Beim Raytracing wird der Schnitt eines Strahls mit diesen Körpern analytisch berechnet. Sollen diese Körper mit Hilfe des z-Buffer-Algorithmus dargestellt oder mit Hilfe des Radiosity-Verfahrens beleuchtet werden, so erfolgt automatisch eine Unterteilung in Polygone, wobei die Facettierung durch das Schlüsselwort `FACET` angegeben wird.

Als vordefinierte Körper existieren `KUGEL`, `WUERFEL`, `ZYLINDER`, `KEGEL` und `TORUS`. Sie werden ohne zusätzliche Schlüsselwörter aufgerufen. In der Beispielszene `Eine_Ganz_Einfache_Szene` ist `KUGEL;` der Aufruf eines vordefinierten Körpers.

A.3.4 CSG-Ausdrücke

In CSG-Ausdrücken können verschiedene Objekte und vordefinierte Körper mit den Operatoren UND, ODER und NICHT kombiniert werden. CSG-Ausdrücke stehen immer in Klammern. Auch sämtliche Teilausdrücke eines CSG-Ausdrucks müssen geklammert werden. Wird ein Objekt, das kein vordefinierter Körper ist, in einem CSG-Ausdruck verwendet, so wird das Schlüsselwort SOLID dem Namen des Objektes vorangestellt.

Es folgen einige Beispiele für gültige CSG-Ausdrücke, wenn Kreuz und Herz definierte Objekte sind:

```
(KUGEL);
(NICHT KUGEL);
(KUGEL FACET 30,10 ODER ZYLINDER);
(KUGEL ODER NICHT ZYLINDER);
(KUGEL ODER SOLID Kreuz);
((NICHT SOLID Herz ODER Zylinder) UND
 (KUGEL UND NICHT SOLID Kreuz));
```

In PRIM werden CSG-Ausdrücke zur Zeit nur ansatzweise behandelt. Sie sind im wesentlichen für eine Erweiterung auf das Raytracing-Verfahren gedacht.

A.3.5 Transformationen und Oberflächeneigenschaften

Am Anfang eines Teilobjektes können Transformationen und Oberflächeneigenschaften definiert werden. Diese Definitionen gelten dann auch für die nächsten Teilobjekte des Objektes, es sei denn, vor diesen werden sie umdefiniert. Ist dies der Fall, so werden die Oberflächeneigenschaften durch die neuen ersetzt, die Transformationen werden mit den alten *verkettet*.

Bei einem Aufruf eines Objektes innerhalb eines anderen geschieht folgendes: Das aufgerufene Objekt (besser: dessen Instanz) erbt die gerade aktuellen Transformationen und Eigenschaften. Wenn innerhalb des aufgerufenen Objektes Teile der Oberflächeneigenschaften umdefiniert werden, so hat dessen Instanz natürlich die in ihm definierten Eigenschaften. Die Transformationen der Instanz werden in diesem Fall mit den Transformationen des aufrufenden Objektes verkettet. Auf diese Weise ist es möglich, daß ein und dasselbe Objekt nicht nur an verschiedenen Stellen erscheint, sondern dort auch jeweils verschiedene Eigenschaften hat.

Hierzu ein Beispiel:

```
GED_VERSION 1;

OBJEKT FlachBall;
  VERSCHIEBE 1.5,0,1.5; // Verschiebe ist eine Transformation
  SCHATTIERUNG FLACH;   // Schattierung Flach ist eine
                        // "Oberflaecheneigenschaft"
  KUGEL FACET 20;
```

```
ENDOBJEKT;

OBJEKT TreeBalls;
  SCHATTIERUNG PHONG;    // Schattierung Phong ist eine
                         // "Oberflaecheneigenschaft"
  KUGEL FACET 20;

  SOLIDMAP MARMOR;       // SolidMap Marmor ist eine
                         // Oberflaecheneigenschaft
  ZEICHNE FlachBall;

  SOLIDMAP HOLZ;         // SolidMap Holz ist eine
                         // Oberflaecheneigenschaft
  SKALIERE TEXTUR 5,5,5;// Skaliere Textur ist eine Transformation
  DREHE TEXTUR X,20;     // Drehe Textur ist eine Transformation
  DREHE Z,120;           // Drehe ist eine Transformation
  ZEICHNE FlachBall;

  DREHE Z,120;           // Drehe ist eine Transformation
  ZEICHNE FlachBall;
ENDOBJEKT;

AMBIENTELQ 1,1,1;
GERICHTETELQ 1,1,1 1,1,-1;  //damit auch 'was zu sehen ist

SZENE VererbungsLehre;
  ZEICHNE TreeBalls;
END.
```

Da in `FlachBall` die Schattierungsart explizit auf `FLACH` gesetzt wird, ist es für die Kugeln in `FlachBall` egal, daß in `TreeBalls` die Schattierungsart auf `PHONG` gestellt ist. Nicht so bei den Texturen. Da `FlachBall` keine Texturen definiert und auch nicht explizit ausschaltet (was mit `NOMAP;` geschehen kann), erbt es die Texturen von `TreeBalls`. Das bedeutet, daß die erste Kugel, die durch `Zeichne FlachBall` erzeugt wird, aus Marmor, die zweite und die dritte aus Holz sind. Die erste Kugel in `TreeBalls` ist Phong-schattiert und weder texturiert noch verschoben.

A.3.6 Teilobjekte im Detail: Polygonobjekte

Polygonobjekte bestehen aus einer Punkt- und einer Flächenliste. Dabei beginnt die Punktliste mit dem Schlüsselwort `PUNKTE` und die Flächenliste mit `FLAECHEN`. Danach folgen jeweils eine Liste von Punkten bzw. eine Liste von Flächen, wobei in der Flächenliste über Indizes auf die in der Punktliste definierten Punkte zugegriffen wird.

Die allgemeine Form für eine Punktdefinition lautet:

```
X-Koordinate, Y-Koordinate, Z-Koordinate
```

```
[TexturX, TexturY [, TexturZ]] [SCHARF];
```

Die Angabe der Texturkoordinaten ist optional. Fehlt sie, so werden die Koordinaten auch als Texturkoordinaten gespeichert.

Das optionale Schlüsselwort SCHARF hat folgende Funktion: Ist ein Punkt scharf, so wird beim Beleuchten des Punktes die Normale der Fläche, deren Eckpunkt der Punkt ist, zur Berechnung des Reflexionsvektors herangezogen. Da der Punkt Eckpunkt von verschiedenen Flächen sein kann, wird er demzufolge mehrfach beleuchtet, jeweils mit dem entsprechenden Normalenvektor der verschiedenen Flächen. Soll z.B. ein Würfel als Polygonobjekt definiert werden (was nicht nötig ist, da er als vordefinierter Körper existiert), so würden die acht Eckpunkte des Würfels alle als scharf definiert.

Der umgekehrte Fall tritt ein, wenn bei einem Punkt das Schlüsselwort SCHARF fehlt. In diesem Fall wird beim Beleuchten des Punktes nur *ein* Normalenvektor verwendet, nämlich der Mittelwert aller Normalen der Flächen, an denen der Punkt als Eckpunkt beteiligt ist. Würde man eine Kugel als Polygonobjekt definieren, so wäre keiner der „Eckpunkte" scharf.

Bei der Radiosity-Berechnung steuert die Angabe von SCHARF, ob für den Punkt die Energiedichten der an ihn angrenzenden Flächen gemittelt werden soll oder nicht. Im zweiten Fall wird bei der Darstellung die Energiedichte der zu zeichenende Fläche als Energiedichte des Punktes angenommen (siehe auch den Abschnitt 18.4, BerechnePunktFarbenAusEnergien).

Die allgemeine Form einer Flächendefinition lautet:

```
Punktindizes [FLIP] [FACET GanzeZahl]
[R 3Vektor] [B 3Vektor] [DB 3Vektor];
```

Punktindizes steht für eine Liste von ganzen Zahlen, jeweils durch Kommata getrennt. Diese Zahlen geben die Nummern der Punkte der vorangehenden Punktliste an, die die Fläche bilden sollen. Der erste Punkt der Punktliste hat die Nummer Null.

Mit dem optionalen Schlüsselwort FLIP kann erreicht werden, daß eine Fläche von beiden Seiten zu sehen ist. Kann man davon ausgehen, daß eine Fläche immer nur von einer Seite sichtbar ist, so sollte FLIP entfallen. Die Renderingzeit ist dann deutlich kürzer. Die Flächen eines Würfels beispielsweise sind nur von einer Seite (von außen) zu sehen, die eines Blattes Papier dagegen von beiden Seiten. Intern wird eine Fläche zweimal abgespeichert, falls FLIP definiert ist. Der Grund ist u.a., daß die beiden korrespondierenden Flächen bei der Radiosityberechnung verschiedene Energien empfangen.

Ist FLIP nicht vorhanden, so wird die Fläche nur dann dargestellt, wenn der Normalenvektor der Fläche zum Betrachter hinzeigt. Der Normalenvektor bildet sich aus dem Kreuzprodukt zweier Kanten der Fläche. Aus diesem Grund ist bei Flächen mit fehlendem FLIP die Reihenfolge der Punktindizes wichtig. Durchläuft man den Rand der Fläche von Punkt zu Punkt, so wird die Richtung des Normalenvektors mit Hilfe der Rechten-Hand-Regel bestimmt.

Ein Beispiel:

```
PUNKTE
  1,0,0;
  0,1,0;
  -1,0,0;
FLAECHEN
  0,1,2;  // Flaechennormale zeigt in positive z-Richtung
  0,2,1;  // Flaechennormale zeigt in negative z-Richtung
```

Durch das Schlüsselwort FACET, gefolgt von einer ganzen Zahl, wird die Aufteilung der Fläche in kleinere Unterflächen erreicht. Dies kann beim Radiosity-Verfahren, beim „normalen" Gouraud-Shading und beim Darstellen texturierter Flächen Vorteile haben. Da aufgrund der Unterflächen mehrere Beleuchtungsberechnungen erfolgen, ist die Abtastung der Lichtverhältnisse höher und demzufolge genauer. Perspektivische Verzerrungen werden durch die Facettierung nicht so stark dargestellt. Für das Raytracing ist die Unterteilung nicht sinnvoll, da hier die Abtastung nicht von der Anzahl der Flächen (oder besser: von der Anzahl der Eckpunkte der Flächen) sondern von der Anzahl der Pixel, die zu setzen sind, abhängt.

FACET muß größer oder gleich 1 sein. FACET 3 bedeutet z.B., daß die kürzeste Kante des entsprechenden Polygons in mindestens 3 Teile aufgeteilt wird. In wieviele Teile das Polygon genau geteilt wird, hängt stark von seiner Form ab. Im allgemeinen wird es so geteilt, daß seine kürzeste Kante in n Teile zerlegt wird, wobei n die Zweierpotenz ist, die größer oder gleich der Zahl hinter FACET ist.

Die Unterflächen erben von den Originalflächen die optionalen Angaben. Die für die neuen Flächen nötigen Eckpunkte einschließlich ihrer Texturkoordinaten werden automatisch erzeugt. Die neuen Punkte sind grundsätzlich nicht scharf.

Die optionalen Angaben [R 3Vektor] und [B 3Vektor] dienen zur Angabe der Reflektivität (R) und der Energiedichte (B) der Fläche. Dabei bezeichnet 3Vektor drei reelle Zahlen, jeweils durch Kommata getrennt. Diese Zahlen entsprechen dem Rot-, Grün- und Blauanteil. Der Bereich von 0 bis 1 entspricht keiner bis voller Reflexion. Fehlt die Reflektivität, so wird stattdessen die aktuelle diffuse Farbe (OD, siehe Beschreibung der Oberflächeneigenschaften weiter unten) gewählt. Im andern Fall ersetzt R für diese eine Fläche OD.

Die Energiedichte B wird bei der Radiosity-Berechnung verwendet. Flächen, deren Energiedichte ungleich Null ist, sind selbstleuchtend, also flächenhafte Lichtquellen.

Verwendet man die Punkte aus dem letzten Beispiel, so ist die Fläche

```
0,1,2 FLIP R 1,0,0 B 20,20,20;
```

ein Dreieck, das von beiden Seiten zu sehen ist (FLIP), das weißes Licht diffus rot reflektiert (R 1,0,0) und eine Energiedichte von (20,20,20) hat. Da die Fläche beidseitig zu sehen ist, wird auch ihre Energie beidseitig ausgestrahlt. Sie emittiert also doppelt so viel Energie, wie die gleiche Fläche, bei der FLIP fehlt.

Gerade in Zusammenhang mit Flächen, für die $B \neq 0$ ist, sollte je nach Größe eine Facettierungsangabe vorgenommen werden, da sonst unter Umständen die Energie der Lichtquelle mit einem einzigen Schuß verteilt wird. Hierdurch könnten scharfe Schattengrenzen entstehen, obwohl aufgrund der Größe der Lichtquelle nur weiche Schatten zu erwarten wären.

A.3.7 Teilobjekte im Detail: Vordefinierte Körper

Im Gegensatz zu polygonalen Körpern wird der Schnitt eines Strahls und eines vordefinierten Körpers im Raytracer analytisch berechnet. Deshalb erscheinen die Konturen solcher Körper beim Raytracing immer perfekt. Beim Rendern mittels z-Buffer und bei der Radiosity-Iteration werden die vordefinierten Körper in Polygone zerlegt. Für vordefinierte Körper in einer GED-Datei ist die Angabe der Facettierung möglich, die bei der Umformung der Körper in Polygone berücksichtigt wird. Die Einschränkungen bei der Facettierung von Polygonen gelten nicht für die vordefinierten Körper. Für sie wird die Facettierung exakt nach den Angaben der GED-Datei vorgenommen.

A.3.7.1 Die Kugel als vordefinierter Körper

Die Syntax für eine Kugel lautet:

```
KUGEL [FACET GanzeZahl [, GanzeZahl]];
```

Der Mittelpunkt der Kugel liegt im Ursprung, ihr Radius beträgt 1. Soll die Kugel an anderer Stelle erscheinen oder einen von 1 verschiedenen Radius besitzen, so muß das mit Hilfe von vorangehenden Transformationen angegeben werden. Die optionale erste ganze Zahl gibt die Facettierung des „Äquators" an, die zweite die Facettierung in „Nord-Süd-Richtung". Die Standardwerte der Facettierung, d.h. die Werte, die gewählt werden, wenn die Angabe fehlt, wird ebenso wie die der andern vordefinierten Körper am Ende dieses Anhangs mit den anderen Standardwerten für GED-Dateien zusammengefaßt.

In Abb. A.2 sind die Drahtmodelle einer Kugel mit verschiedenen Facettierungen zu sehen. Das dazugehörige Progammstück ist:

```
KUGEL;
KUGEL FACET 20,5;
KUGEL FACET 20,20;
```

A.3.7.2 Der Würfel als vordefinierter Körper

Die Syntax für einen Würfel lautet:

```
WUERFEL [FACET GanzeZahl];
```

Der Würfel hat seinen Schwerpunkt im Ursprung und eine Kantenlänge von 2. Er deckt in x-, y- und z-Richtung jeweils den Bereich von -1 bis 1 ab. Die optionale ganze Zahl gibt an, wie fein die sechs Flächen des Würfels unterteilt werden sollen. Ist sie z.B. 4, so besteht jede Fläche aus 4×4 Unterflächen (Abb. A.3).

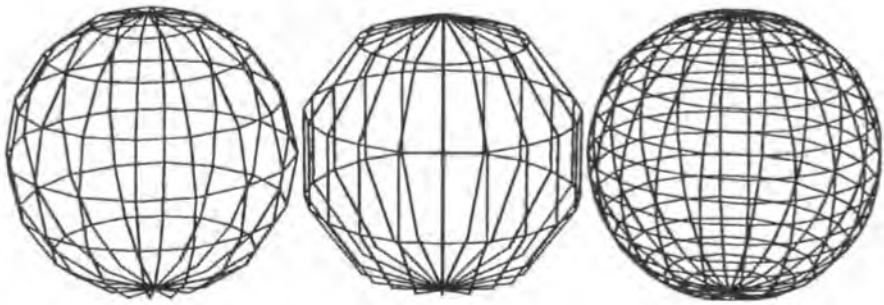


Abb. A.2 Verschieden facettierte Kugeln. Die linke Kugel ist mit der Standardeinstellung, die mittlere mit 20 Facetten in horizontaler und 5 Facetten in vertikaler und die rechte mit 20 Facetten in horizontaler und vertikaler Richtung dargestellt

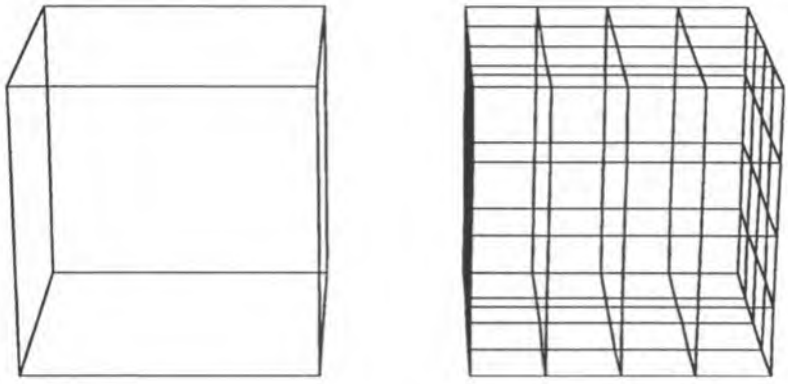


Abb. A.3 Verschieden facettierte Würfel. Der linke Würfel ist ohne Facettenangabe definiert, der rechte mit 4 Facetten in jeder Raumrichtung

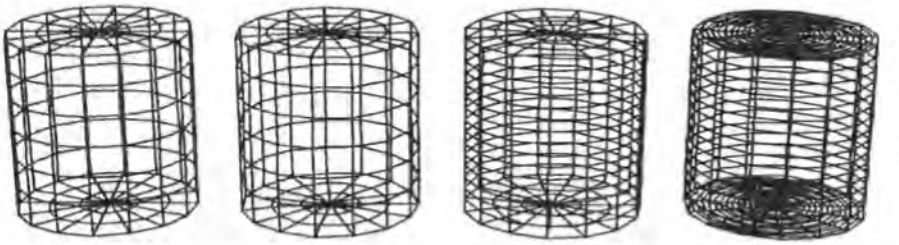


Abb. A.4 Verschieden facettierte Zylinder

A.3.7.3 Der Zylinder als vordefinierter Körper

Die Syntax für einen Zylinder lautet:

```
ZYLINDER [FACET GanzeZahl [, GanzeZahl [, GanzeZahl ]]];
```

Der Zylinder hat seinen Schwerpunkt im Ursprung, einen Radius von 1 und eine Höhe von 2. Die erste optionale Zahl gibt an, in wieviele „Tortenstücke" der Zylinder zerlegt werden soll, die zweite, in wieviele Scheiben und die dritte, in wieviele konzentrische Ringe er zerlegt werden soll. Abb. A.4 zeigt Zylinder verschiedener Facettierung, die mit dem Programmstück

```
ZYLINDER;
ZYLINDER FACET 15;
ZYLINDER FACET 15,15;
ZYLINDER FACET 15,15,15;
```

erzeugt wurden.

A.3.7.4 Der Kegel als vordefinierter Körper

Die Syntax für einen Kegel lautet:

```
KEGEL [FACET GanzeZahl [, GanzeZahl [, GanzeZahl ]]];
```

Die Grundplatte befindet sich in der $(z = -1)$-Ebene und hat einen Radius von 1. Die Höhe des Kegels beträgt 2. Die optionalen Parameter haben die gleiche Bedeutung wie die des Zylinders. In Abb. A.5 sind Kegel mit der folgenden Definition dargestellt:

```
KEGEL;
KEGEL FACET 15;
KEGEL FACET 15,15;
KEGEL FACET 15,15,15;
```

A.3.7.5 Der Torus als vordefinierter Körper

Die Syntax für einen Torus lautet:

```
TORUS ReelleZahl [FACET GanzeZahl [, GanzeZahl ]] ;
```

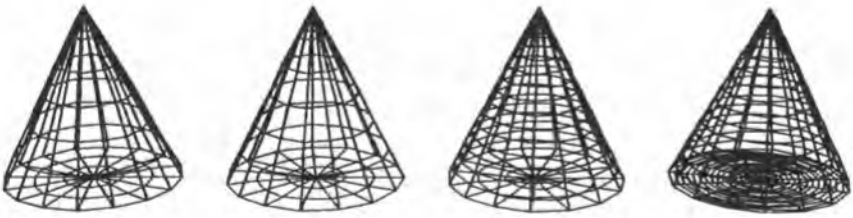


Abb. A.5 Verschieden facettierte Kegel

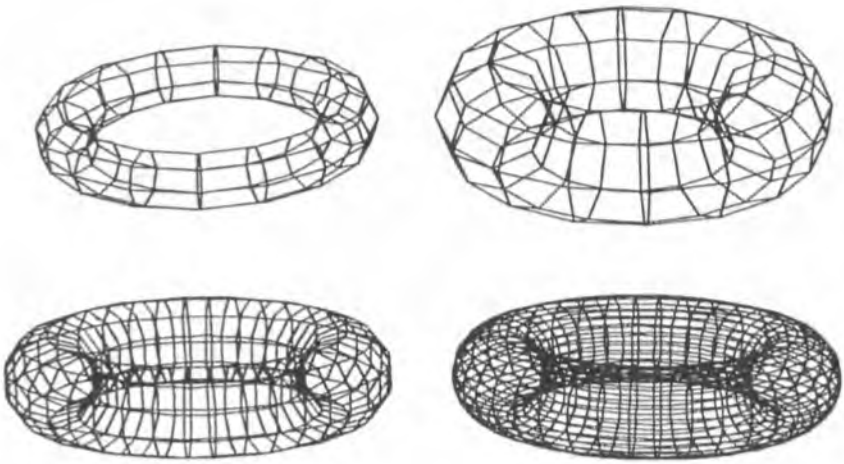


Abb. A.6 Verschieden facettierte Tori

Im Gegensatz zu den anderen Grundkörpern muß beim Torus immer eine reelle Zahl angegeben werden. Sie entspricht dem Verhältnis zwischen kleinem und großem Radius des Torus. Er liegt in der $z = 0$-Ebene. Der große Radius ist 1. Der erste optionale Parameter gibt die Unterteilung des großen, der zweite die Unterteilung des kleinen Radius an. In Abb. A.6 ist das zum folgenden Listing gehörige Bild dargestellt.

```
TORUS 0.2;
TORUS 0.4;
TORUS 0.3 FACET 40,10;
TORUS 0.3 FACET 40,20;
```

A.4 Transformationen

Transformationen (und auch Oberflächeneigenschaften) können vor den verschiedenen Arten von Teilobjekten stehen. Auch nach dem letzten Teilobjekt können diese Angaben gemacht werden, die dann allerdings ignoriert werden.

Es stehen folgende Befehle für die Transformation zur Verfügung:

```
VERSCHIEBE [[ MIT ] TEXTUR ] 3Vektor;
DREHE      [[ MIT ] TEXTUR ] X | Y | Z ReelleZahl ;
SKALIERE   [[ MIT ] TEXTUR ] 3Vektor | ReelleZahl ;
```

Diese Befehle können beliebig häufig vor einem Teilobjekt stehen. Einige Beispiele:

```
1) VERSCHIEBE 0,2,3;
2) VERSCHIEBE TEXTUR 3,2,-1.5;
3) VERSCHIEBE MIT TEXTUR 2,3,4;

4) DREHE x,60;
5) DREHE TEXTUR y,13;

6) SKALIERE 2;
7) SKALIERE MIT TEXTUR 2,-3,4;
```

Im ersten Beispiel werden alle Teilobjekte, die hinter dieser Stelle in einem Objekt stehen, um 0 in x-, 2 in y- und 3 in z-Richtung verschoben. Das zweite Beispiel verschiebt dagegen nicht die Teilobjekte, sondern die Textur auf den Teilobjekten. Das dritte Beispiel verschiebt sowohl die Teilobjekte als auch ihre Texturen, so daß die Texturen relativ zu den Teilobjekten ihre Position behalten.

Beispiel 4 dreht die Teilobjekte um die x-Achse um 60 Grad, Beispiel 5 die Textur der Teilobjekte um die y-Achse um 13 Grad.

Beispiel 6 skaliert die Teilobjekte in alle Richtungen um den Faktor 2, Beispiel 7 skaliert die Teilobjekte einschließlich der Textur um 2 in x-Richtung, -3 in y-Richtung und 4 in z-Richtung.

Beim Einlesen von GED-Dateien werden alle Transformationen, die vor einem Teilobjekt stehen, zu einer einzigen Matrix zusammengefaßt. Diese Matrix beschreibt dann die Transformation vom Anfang des Objektes bis zu diesem Teilobjekt.

A.5 Oberflächeneigenschaften

Das Aussehen von Teilobjekten kann durch diverse Oberflächeneigenschaften verändert werden. Im einzelnen sind dies:

- der ambiente Reflexionskoeffizient KA,
- der diffuse Reflexionskoeffizient KD,
- der spiegelnde Reflexionskoeffizient KS,

- der Exponent für die spiegelnde Reflexion `OE`,
- die Transparenz `T`,
- der Brechungsindex `N`,
- die diffuse Farbe `OD`,
- die spiegelnde Farbe `OS`,
- die Art der Schattierung und
- die Art der Textur.

Für die einzelnen Eigenschaften gilt folgende Syntax:

```
KA ReelleZahl ;
KD ReelleZahl ;
KS ReelleZahl ;
OE ReelleZahl ;
T  ReelleZahl ;
N  ReelleZahl ;
OD 3Vektor ;
OS 3Vektor ;

SCHATTIERUNG FLACH | GOURAUD | PHONG ;
```

Bei der Berechnung des Bildes durch Raytracing würde die Angabe der Schattierungsart nicht berücksichtigt. Die Beleuchtungsberechnung erfolgt bei diesem Verfahren für jeden vom Sichtstrahl geschnittenen Punkt.

Für die Textur sind folgende Möglichkeiten gegeben:

```
MAP       Dateiname ;
BUMPMAP   Dateiname ;
SOLIDMAP HOLZ ;
SOLIDMAP MARMOR ;
NOMAP ;
```

Hierbei entspricht `MAP` dem Mappen einer zweidimensionalen Textur, also dem Kleben eines Bildes auf ein Objekt, `BUMPMAP` dem Bumpmapping und `SOLIDMAP` einer 3D-Textur. Das Format der Texturdateien für `MAP` und `BUMPMAP` wird weiter unten beschrieben. Unabhängig von der Schattierungsart werden Teilobjekte, die mit einer Bump-Textur versehen sind, Phong-schattiert. Eine Bump-Textur verändert die Normalen innerhalb einer Fläche. Daher würde sich diese Textur bei anderen Schattierungsverfahren nicht bemerkbar machen.

`HOLZ` und `MARMOR` sind prozedurale Texturen. Einer Erweiterung der Syntax und des Programmes auf andere prozedurale Texturen steht natürlich nichts im Weg. `NOMAP` hat zur Folge, daß für die folgenden Teilobjekte keine Textur verwendet wird. Um den Programmieraufwand nicht ins Uferlose wachsen zu lassen, wird pro Teilobjekt maximal eine Textur (bei mehreren Angaben ist es die letzte) berücksichtigt.

A.6 Gültigkeitsbereich von Transformationen und Oberflächeneigenschaften

Der Gültigkeitsbereich von Transformationen und Oberflächeneigenschaften wurde bereits angesprochen. Beide gelten innerhalb eines Objektes ab der Stelle, an der sie definiert sind. Oberflächeneigenschaften werden an Instanzen (Aufrufe) anderer Objekte vererbt und behalten dort ihre Gültigkeit, es sei denn, sie werden explizit umdefiniert. Transformationen werden verkettet.

Soll der Zustand der Transformationen und/oder der Oberflächeneigenschaften an einer bestimmten Stelle gerettet werden, um ihn an anderer Stelle wiederherzustellen, so können die beiden Befehle PUSH und POP verwendet werden, wobei zu jedem PUSH im selben Objekt genau ein korrespondierendes POP definiert sein muß.

Die Syntax für PUSH und POP lautet:

```
PUSH was ;
POP  was ;
```

Wobei was für eine Liste von

```
TEXTURKOORDINATEN, KOORDINATEN, EIGENSCHAFTEN
```

steht. Die Trennung der Einträge erfolgt durch Kommata. Ist die Liste leer, so werden sowohl die Koordinaten als auch die Texturkoordinaten und die Eigenschaften gerettet bzw. wiederhergestellt.

Mit Hilfe von PUSH und POP ist es möglich, innerhalb eines Objektes bestimmte Aktionen lokal zu halten, ohne daß für diese Aktionen ein eigenes Objekt definiert werden muß. Tatsächlich ist der Aufruf eines Objektes B innnerhalb des Objektes A identisch mit der Befehlsfolge

```
.       // Befehle von A
PUSH;
 .      // Befehle von B
POP;
.       // weiter mit Befehlen von A
```

Damit sind sämtliche Möglichkeiten der Syntax einer GED-Datei angesprochen, die sich auf Objekte und – als speziellem Objekt – auf die Szene beziehen.

A.7 Angaben direkt vor der Szene: Betrachter- und Lichtquelleninformationen

Zwischen dem letzten Objekt einer GED-Datei und der Szene (bzw. zwischen der Versionsangabe und der Szene, wenn keine Objekte definiert sind) können noch Angaben über den Betrachterstandpunkt, die Blickrichtung u.ä., gefolgt von Angaben über Lichtquellen, die die Szene beleuchten, gemacht werden. Alle Angaben sind optional. Fehlen sie, so werden sie auf Standardwerte gesetzt. Es

empfiehlt sich aber, auf jeden Fall mindestens eine Lichtquelle zu definieren, da man sonst außer schwarzen Konturen der definierten Objekte vor dem Hintergrund nichts sieht.

A.7.1 Angaben zum Betrachter

Folgende Angaben zum Betrachter sind möglich:

```
AUGE           3Vektor ;
REFERENZPUNKT  3Vektor ;
UPVEKTOR       3Vektor ;
BILDABSTAND    ReelleZahl ;
VORDERCLIPEBENE ReelleZahl ;
HINTERCLIPEBENE ReelleZahl ;
BILDBREITE      ReelleZahl ;
BILDSCHIRMFENSTER GanzeZahl, GanzeZahl,
                  GanzeZahl, GanzeZahl ;
PIXELRATIO        ReelleZahl ;
```

AUGE legt die Position des Auges fest, REFERENZPUNKT den Punkt, auf den AUGE schaut. Die Projektion von REFERENZPUNKT auf den Bildschirm ist immer der Mittelpunkt des Bildschirms. UPVEKTOR gibt an, welche Richtung für den Betrachter „oben" ist. Mit BILDABSTAND kann angegeben werden, wie weit die Bildebene vom AUGE entfernt ist. Je kleiner BILDABSTAND ist, umso größer ist die perspektivische Verzerrung.

VORDERCLIPEBENE gibt an, in welchem Abstand von der Bildebene Objekte abgeschnitten werden. VORDERCLIPEBENE muß kleiner als BILDABSTAND sein, da sonst Objekte dargestellt würden, die hinter dem Auge liegen, was bei der Projektion zu eigenartigen Effekten führt. Objekte, die hinter der HINTERCLIPEBENE liegen, werden ebenfalls abgeschnitten. Dies ist speziell für den z-Buffer-Algorithmus nötig, da dort der Bereich zwischen VORDERCLIPEBENE und HINTERCLIPEBENE auf den maximalen z-Buffer-Bereich abgebildet wird. HINTERCLIPEBENE muß kleiner sein als VORDERCLIPEBENE. Soll HINTERCLIPEBENE vom Auge gesehen hinter der Bildebene liegen, so ist ihr Wert negativ.

BILDBREITE gibt an, wie breit der Bereich der Bildebene sein soll, der auf den Bildschirm projiziert wird. Läßt man alle Angaben zum Betrachter konstant und ändert nur BILDBREITE, so entspricht das Verkleinern von BILDBREITE dem Heranzoomen bei einem Teleobjektiv.

Im Gegensatz zu allen übrigen Angaben zum Betrachter, die die Situation im Weltkoordinatensystem beschreiben, sind die Werte für BILDSCHIRMFENSTER in Pixeln angegeben. Die ersten beiden Zahlen bezeichnen die linke obere Ecke des Fensters, in dem die Bildebene dargestellt werden soll, die beiden anderen die Breite und die Höhe. Die letzte Angabe, PIXELRATIO, entspricht dem Verhältnis zwischen Länge und Breite eines Pixels. Diese Angabe ist nötig, damit Grafiken unverzerrt dargestellt werden können. Stimmt dieser Wert nicht, so erscheinen z.B. Kugeln nicht rund. Um PIXELRATIO zu bestimmen, eignen sich ein Lineal und die einfache Anwendung des Dreisatzes hervorragend.

A.7.2 Angaben zu den Lichtquellen

Hinter den Angaben zum Betrachter können (und sollten) Angaben zu den Lichtquellen folgen. Als Lichtquellen sind ambiente, gerichtete, punktförmige und Strahler möglich (siehe Abschnitt 10.2).

Die Syntax lautet:

```
AMBIENTELQ    3Vektor ;
GERICHTETELQ  3Vektor 3Vektor ;
PUNKTLQ       3Vektor 3Vektor 2Vektor ;
STRAHLER      3Vektor 3Vektor 3Vektor 2Vektor
              [ReelleZahl [ReelleZahl]];
```

Der `3Vektor` bei `AMBIENTELQ` definiert die Farbe der Lichtquelle.

Der erste `3Vektor` bei `GERICHTETELQ` definiert die Farbe, der zweite die Richtung, in der das Licht ausgestrahlt wird.

Bei der `PUNKTLQ` bezeichnet der erste `3Vektor` die Farbe und der zweite die Position. Der `2Vektor` (zwei Zahlen, durch Komma getrennt) definiert die beiden Abklingkoeffizienten.

Die Parameter von `STRAHLER` haben der Reihe nach folgende Bedeutung: Farbe, Richtung, Position, Abklingkoeffizienten, Konzentrationsexponent und Öffnungswinkel in Grad. Die Angabe von Exponent und Öffnungswinkel sind optional.

A.8 Hintergrund

Letztes Sprachelement von `GED` ist das Hintergrundobjekt. Es dient dazu, im Bildhintergrund eine Textur oder einen Farbverlauf darzustellen. Das Hintergrundobjekt ist im Prinzip ein Teilobjekt mit besonderen Einschränkungen. Es darf nur in der Szene definiert sein und muß dort als erstes Teilobjekt stehen. Vor dem Hintergrund sind aber, wie bei allen Teilobjekten, Angaben zur Transformation und zu den Oberflächeneigenschaften möglich. Hiermit kann im Hintergrund durch die Angabe einer Textur ein Bild dargestellt werden, wobei Lage und Größe des Bildes durch Transformation der Textur verändert werden können.

Die Syntax für ein Hintergrundobjekt lautet:

```
HINTERGRUND [3Vektor 3Vektor] ;
```

Die optionalen Vektoren geben die Farbe des Hintergrundes an der oberen und der unteren Kante an. Zwischen den Kanten wird linear interpoliert.

Drei Beispiele:

```
...   //Definition von "Puppen" etc.
SZENE Hintergrund_Demo;
  MAP ..\textures\holz.bmp;
  HINTERGRUND;
```

```
  ZEICHNE Puppen;
END.

...   //Definition von "Puppen" etc.
SZENE Hintergrund_Demo;
  MAP ..\textures\holz.bmp;
  HINTERGRUND 0.3,0.3,0.3 1,1,1;
  ZEICHNE Puppen;
END.

...   //Definition von "Puppen" etc.
SZENE Hintergrund_Demo;
  SKALIERE Textur 4;
  MAP ..\textures\holz.bmp;
  HINTERGRUND 0.3,0.3,0.3 1,1,1;
  ZEICHNE Puppen;
END.
```

Die drei Programme erzeugen die in Abb. XI des farbigen Bildteils dargestellten Bilder.

Hiermit sind sämtliche Sprachelemente einer GED-Datei beschrieben. Die Syntax kann und sollte bei Bedarf den speziellen Anforderungen der Leserin bzw. des Lesers angepaßt werden. Hierfür sind folgende Schritte nötig:

1. Definition der neuen (bzw. erweiterten Syntax)
2. Änderung der betroffenen Datenstrukturen auf die neuen Möglichkeiten
3. Implementierung der Syntax in PRIMREAD.PAS
4. Implementierung der Möglichkeiten in den betroffenen Rendering-Programmen aufgrund der veränderten Datenstrukturen.

In den zwei Tabellen A.1 und A.2 sind die vordefinierten Werte innerhalb der grafischen Eingabedatei zusammengefaßt. Die erwähnten Größen XMAX und YMAX hängen von der Auflösung der verwendeten Grafikkarte ab.

Tabelle A.1 Vordefinierte Werte in GED

Sprachelement	Wert
KA	0.33333
KD	0.33333
KS	0.33333
OE	1
T	0
OD	1,1,1
OS	1,1,1
SCHATTIERUNG	PHONG
AUGE	7,-30,10
REFERENZPUNKT	0,0,0
UPVEKTOR	0,0,1
BILDABSTAND	20
VORDERCLIPEBENE	10
HINTERCLIPEBENE	-10
BILDBREITE	10
BILDSCHIRMFENSTER	0,0,XMAX,YMAX
PIXELRATIO	1
HINTERGRUND	$(0, 0.1, 0.2), (0, 0.2, 0.6)$

Tabelle A.2 Vordefinierte Werte optionaler Parameter in GED

Sprachelement	Optionaler Parameter 1	Optionaler Parameter 2	Optionaler Parameter 3
Kugel	16	8	–
Wuerfel	1	–	–
Zylinder	16	6	3
Kegel	16	6	3
Torus	16	8	–
Strahler	3	20	–

A.9 Die von LeseGEDDatei erzeugten Datenstrukturen

In diesem Abschnitt werden die Module PRIMGLOB, PRIMREAD und PRIMDERE beschrieben. Sie bestimmen, wie eine GED-Datei nach dem Einlesen im Speicher steht.

Wie schon an mehreren Stellen des Buches erwähnt, liest das Hauptprogramm eine Datei ein, die die darzustellende Szene beschreibt. Die Syntax der Eingabedateien wurde bereits oben besprochen. Dieser Teil des Anhangs befaßt sich nun damit, welche Form die entprechenden internen Datenstrukturen nach dem Einlesen der Dateien haben.

In PRIMGLOB sind die Typen der erzeugten Strukturen definiert. Diese Typen sind für alle weiteren verwendeten Module bekannt, so daß alle Units mit den erzeugten Datenstrukturen arbeiten können. Das eigentliche Einlesen der GED-Datei übernimmt das Modul PRIMREAD. Ergebnis der Prozedur dieser Unit ist im wesentlichen ein Zeiger, an dem quasi sämtliche gelesene Information „hängt".

PRIMDERE erzeugt dann aus diesem Datenbaum eine Flächen-, eine Punkt- und eine Eigenschaftsliste. Bei der Radiosity-Berechnung sowie beim Rendern mit Hilfe des z-Buffers werden dann ausschließlich diese Listen benutzt. Soll ein Raytracing integriert werden, so kann aber ebenfalls der Datenbaum (z.B. für CSG-Objekte) verwendet werden.

A.9.1 Die Unit PRIMGLOB

Die in PRIMGLOB definierten Typen haben zum großen Teil ihre Entsprechung in einem Sprachelement von GED. Diese Entsprechungen sind in Tabelle A.3 zusammengefaßt.

In den folgenden Abschnitten wird auf jeden der in PRIMGLOB definierten Typen kurz eingegangen. Die Reihenfolge der Auflistung entspricht einer „Top-Down" Reihenfolge, da es dann einfacher ist, die implementierten Strukturen nachzuvollziehen. In PRIMGLOB selbst ist die Reihenfolge eher „Bottom-Up", da es in Pascal nicht erlaubt ist, Typen zu benutzen, wenn sie noch nicht definiert sind. Die einzige Ausnahme hiervon sind Zeiger.

A.9.1.1 Zeigertypen von PRIMGLOB

Als Zeiger sind in PRIMGLOB folgende Typen definiert.

```
TeilObjektPtrTYP      = ^TeilObjektTYP;
ObjektPtrTYP          = ^ObjektTYP;
SolidDefinitionPtrTYP = ^SolidDefinitionTYP;
SolidTermPtrTYP       = ^SolidTermTYP;
SolidAusdruckPtrTYP   = ^SolidAusdruckTYP;
LichtquellenPtrTYP    = ^LichtquellenTYP;
EigenschaftPtrTYP     = ^EigenschaftTYP;
GlobaleFlaecheAdrTYP  = LONGINT;
FlaecheAdrTYP         = LONGINT;
```

Tabelle A.3 Entsprechung von Sprachelementen von GED und den verwendeten Typen

Sprachelement	Abschnitt	entsprechende Typen
Objektliste	A.1, A.2	ObjektTYP
Betrachter	A.1, A.7, A.7.1	BetrachterTYP
Lichtquellen	A.1, A.7, A.7.2	LichtquellenTYP AmbienteLqTYP GerichteteLqTYP PunktLqTYP StrahlerTYP
Hintergrundbeschreibung	A.1, A.8	HintergrundTYP
Teilobjektliste	A.2, A.3	TeilObjektTYP GeometrieDatenTYP
Objekt- bzw. Szenenname	A.2	ObjektTYP.Name
Polygonale Teilobjekte:	A.3.1, A.3.6	GeometrieDatenTYP.po
Punktliste	A.3.1, A.3.6	PunktTYP Punkt3DTYP Vektor3DTYP Vektor4DTYP FarbVektorTYP
Flächenliste	A.3.1, A.3.6	FlaecheTYP GlobaleFlaecheTYP ErwFlaechenTYP PolygonTYP
Referenzen auf Objekte	A.3.2	GeometrieDatenTYP.dr
CSG-Ausdrücke	A.3.4	SolidTermTYP SolidAusdruckTYP SolidDefinitionTYP GeometrieDatenTYP.so
Transformationen	A.3.5, A.4, A.6	Mat4TYP
Oberflächeneigenschaften	A.3.5, A.5, A.6	EigenschaftTYP ErwEigenschaftTYP TexturTYP
Vordefinierte Körper	A.3.3, A.3.7	SolidDefinitionTYP
Kugel	A.3.7.1	SolidDefinitionTYP.Ku
Würfel	A.3.7.2	SolidDefinitionTYP.Wu
Zylinder	A.3.7.3	SolidDefinitionTYP.Zy
Kegel	A.3.7.4	SolidDefinitionTYP.Ke
Torus	A.3.7.5	SolidDefinitionTYP.Tor

```
PunktAdrTYP          = LONGINT;
```

Eine Erläuterung erübrigt sich für alle bis auf die letzten drei Typen, da die Strukturen, auf die sie zeigen, weiter unten genauer erklärt werden. GlobaleFlaecheAdrTYP, FlaecheAdrTYP und PunktAdrTYP sind keine Pascal-Zeiger im eigentlichen Sinne. Sie geben vielmehr eine Adresse an, mit deren Hilfe Routinen der Unit PRIMMEM auf die entsprechenden Datenstrukturen zugreifen können (siehe Anhang C).

A.9.1.2 Abspeichern von Objekten: ObjektTYP und andere

```
ObjektTYP        = RECORD
                     Name            : STRING;
                     TeilObjektListe : TeilObjektPtrTYP;
                     next            : ObjektPtrTYP;
                   END;
TeilObjektTYP    = RECORD
                     Eigenschaft     : ErwEigenschaftTYP;
                     GeometrieDaten  : GeometrieDatenTYP;
                     next            : TeilObjektPtrTYP;
                   END;
GeometrieDatenTYP =
     RECORD CASE ArtDerGeometrieDaten: ArtDerGeometrieDatenTYP OF
       Solid          : (so: SolidTermPtrTYP);
       PolygonObjekt  : (po: RECORD
                               PunktAnzahl   :LONGINT;
                               FlaechenAnzahl:LONGINT;
                               Punkte  : PunktAdrTYP;
                               Flaechen: GlobaleFlaecheAdrTYP;
                             END);
       ZeichneReferenz : (dr: ObjektPtrTYP);
     END;
```

ObjektTYP (bzw. ObjektPtrTYP) ist der Typ, über den nach dem Einlesen einer GED-Datei auf alle benutzten Objekte der Datei zugegriffen wird. Die SZENE der GED-Datei ist nichts anderes als eine Variable vom Typ ObjektTYP. Beim Einlesen wird eine verkettete Liste von Objekten erzeugt und die aufrufende Routine erhält den Zeiger auf das „Szenen-Objekt".

ObjektTYP.Name entspricht dem Namen eines Objektes, ObjektTYP.TeilObjektListe beinhaltet in einer verketteten Liste alle Teilobjekte des Objekts. ObjektTYP.next dient während des Einlesens zum Verketten. Der next-Zeiger des „Szenen-Objektes" ist allerdings immer NIL.

Die Teilobjekte eines Objekts haben den Typ TeilObjektTYP. Sie sind ebenfalls als verkettete Liste gespeichert, wobei jedes Objekt auf das erste Teilobjekt Zugriff hat. Ein Teilobjekt selbst besteht einerseits aus den Eigenschaften und andererseits aus den eigentlichen GeometrieDaten (siehe A.3.5). Die Geometriedaten (GeometrieDatenTYP) können je nach Art des Teilobjektes verschieden

sein. Dies ist mit Hilfe eines varianten Feldes mit den Varianten `so`, `po` und `dr` realisiert, wobei `ArtDerGeometrieDaten` angibt, um welche Variante es sich handelt.

`Solid` entspricht einem vordefinierten Körper (Abschnitt A.3.7) oder einem geklammerten CSG-Ausdruck (Abschnitt A.3.4), `PolygonObjekt` einer Punkt- und Flächenliste (Abschnitt A.3.6) und `ZeichneReferenz` der Referenz auf ein Objekt (Abschnitt A.3.2).

A.9.1.3 Abspeichern des Betrachters: `BetrachterTYP`

```
BetrachterTYP    = RECORD
                     Auge,ReferenzPunkt,UPVektor : Vektor3DTYP;
                     BildAbstand,VorderClipEbene,
                     HinterClipEbene,BildBreite  : SINGLE;
                     DCx,DCy,DCw,DCh             : WORD;
                     PixelRatio                  : SINGLE
                   END;
```

Informationen zum Betrachter werden beim Einlesen getrennt gespeichert und ebenfalls getrennt dem aufrufenden Programm übergeben. Bis auf `DCx`, `DCy`, `DCw` und `DCh` sind die Elementnamen von `BetrachterTYP` identisch mit den entprechenden Angaben der GED-Datei (Abschnitt A.7.1). `DCx`, `DCy`, `DCw` und `DCh` sind die Angaben zum Bildschirmfenster.

A.9.1.4 Abspeichern von Lichtquellen: `LichtquellenTYP`

Genauso wie der Betrachter werden auch die Lichtquellen in einer Extrastruktur gespeichert und gesondert an das aufrufende Programm übergeben. Alle Lichtquellen sind in einer verketteten Liste gespeichert. Die Elemente der Liste sind vom Typ `LichtquellenTYP`, wobei je nach `Art` der Lichtquelle eine der vier Varianten gültig ist. `NichtVerdeckt` kann bei einer Raytracing-Implementierung zum Erzeugen von Schatten benutzt werden. Nach dem Einlesen ist `NichtVerdeckt` für alle Lichtquellen auf `TRUE`.

Je nach Art der Lichtquelle folgen dann die nötigen Angaben, die direkt aus der GED-Datei gelesen werden (Abschnitt A.7.2). Deren genaue Beschreibung erfolgte bereits in Kap. 12, der Beschreibung der Beleuchtungsberechnung.

A.9.1.5 Abspeichern des Hintergrundes: `HintergrundTYP`

```
HintergrundTYP   = RECORD
                     FarbeOben           : FarbVektorTYP;
                     FarbeUnten          : FarbVektorTYP;
                     TexturTransformation : Mat4TYP;
                     Textur              : TexturTYP;
                   END;
```

HintergrundTYP ist der letzte Typ, der direkt an das aufrufende Programm geliefert wird. In ihm sind alle Informationen, die für den Hintergrund relevant sind, gespeichert (Abschnitt A.8). Die verschiedenen Transformationen der Textur, die vor der Angabe eines Hintergrundes in der GED-Datei möglich sind, werden bereits beim Einlesen zu einer Matrix TexturTransformation zusammengefaßt. TexturTYP wird im Zusammenhang mit Oberflächeneigenschaften genauer erläutert.

A.9.1.6 Abspeichern von polygonalen Teilobjekten: ErwFlaechenTYP, FlaecheTYP und PunktTYP

```
ErwFlaechenTYP   = RECORD
                     dBA             : FarbVektorTYP;
                     FF              : SINGLE;
                     B               : FarbVektorTYP;
                     Normale         : Vektor3DTYP;
                     PunktAnzahl     : BYTE;
                     FacetZahl       : WORD;
                     Reflektivitaet  : FarbVektorTYP;
                   END;
FlaecheTYP       = RECORD
                     Erw             : ErwFlaechenTYP;
                     PunktIndizes    : VierIndizesTYP;
                     NachbarIndizes  : VierIndizesTYP;
                     ReflektivitaetGueltig:BOOLEAN;
                   END;
PunktTYP         = RECORD
                     Koordinate          : Vektor3DTYP;
                     Normale             : Vektor3DTYP;
                     TexturKoordinate    : Vektor3DTYP;
                     KoordinateDC        : Vektor3DTYP;
                     IstScharf           : BOOLEAN;
                     KoordinateNPC       : Vektor4DTYP;
                     ClipCode            : BYTE;
                     EigenschaftsRef     : EigenschaftPtrTYP;
                     DiffFarbe,SpekFarbe : FarbVektorTYP;
                     TexKoordinateLokal  : BOOLEAN;
                     benutzt             : BOOLEAN;
                   END;
```

Die Typen FlaecheTYP und ErwFlaecheTYP dienen zum Speichern von Flächen, PunktTYP zum Speichern von Punkten polygonaler Teilobjekte einer GED-Datei (Abschnitt A.3.6). Beim Einlesen wird für jedes dieser Teilobjekte eine Punkt- und eine Flächenliste erzeugt, in der die entsprechenden Informationen hintereinander abgelegt werden. Dabei erfolgt die Abspeicherung nicht, wie z.B. bei den Lichtquellen, in einer durch Zeiger verketteten Liste, sondern im XMS-Speicher „am Stück". Die Informationen liegen dann im Speicher direkt hintereinander (siehe auch NewMem, SetMem und GetMem der Unit PRIMMEM). Bei-

spielsweise wird auf den i-ten Punkt der Liste aufgrund einer Basisadresse und `i * sizeof(PunktTYP)` zugegriffen.
Dieses Vorgehen hat zwei Vorteile:

1. Durch die Verwendung des XMS-Speichers kann die Anzahl der Punkte und Flächen sehr hoch sein, die PC-typische 640-kByte-Grenze ist also aufgehoben.
2. Man kann sehr schnell auf das i-te Element der Punkt- bzw. Flächenliste zugreifen, ohne erst alle $(i-1)$ vorangehenden Elemente der Liste zu durchlaufen.

Die Aufspaltung der Flächeninformationen in `FlaechenTYP` und `ErwFlaechenTYP` (`Erw` steht für „Erweiterung") wird im Zusammenhang mit `GlobaleFlaecheTYP` deutlich. Beim Dereferenzieren der eingelesenen Dateien werden nämlich die Flächeninformationen, die in `FlaecheTYP.Erw` stehen, komplett kopiert. Die anderen Informationen aus `FlaecheTYP` werden dagegen gar nicht oder nur verändert übernommen.

`dBA`, `FF` und `B` sind Einträge für die Radiosity-Berechnung. Es ist möglich, direkt in der Eingabedatei für Flächen die Reflektivität `R` und die Energiedichte `B` anzugeben (siehe Seite 333). Die Speicherung erfolgt an den entsprechenden Stellen von `FlaecheTYP.Erw`. In `FlaecheTYP.Erw.dBA` wird zunächst nur `B` abgespeichert. Die Multiplikation mit dem Flächeninhalt, durch die `B` zu `dBA` wird, erfolgt erst beim Dereferenzieren, da erst dann die echte Flächengröße bekannt ist. Durch die Transformation von Objekten vor ihrem Aufruf kann sich die Ausdehnung der Objekte ändern.

Aus Gründen, die im Zusammenhang mit `GlobaleFlaecheTYP` näher erläutert werden, *muß* `dBA` der erste und `FF` der zweite Eintrag in `ErwFlaecheTYP` sein.

Die `Normale` der Fläche wird nach dem Einlesen des Teilobjektes automatisch errechnet. Hierzu dient das normierte Kreuzprodukt zweier Kanten der Fläche. `PunktAnzahl` gibt an, wieviele Eckpunkte die Fläche hat. Da direkt beim Einlesen Flächen mit mehr als vier Eckpunkten in Drei- und Vierecke zerlegt werden, ist `PunktAnzahl` immer 3 oder 4.

`FacetZahl` speichert die Angabe zur Facettierung der Fläche (siehe Seite 333). Wenn in der `GED`-Datei für eine Fläche keine Angaben zur Facettierung der Fläche vorhanden sind, so ist `FacetZahl` $= 1$. Die Unterteilung von Flächen mit `FacetZahl` $\neq 1$ übernimmt `ErzeugePatchesUndElemente` (Abschnitt 18.4.1).

Zusätzlich zu `Erw` sind in `FlaecheTYP` die Indizes der Eckpunkte der Fläche (`PunktIndizes`) sowie die Indizes der Nachbarflächen (`NachbarIndizes`) gespeichert. Die `NachbarIndizes` werden beim Einlesen generiert. Zwei Flächen gelten dann als benachbart, wenn sie sich zwei Punkte in umgekehrter Reihenfolge an einer Kante teilen, von denen mindestens einer nicht „scharf" ist oder wenn beide Flächen den gleichen Normalenvektor haben.

Die Zuordnung von Punkt- und Nachbarindizes ist nicht unwesentlich. Sie wird bei der Unterteilung in Patches und Elemente ausgenutzt. Für die Zuordnung gilt: Der i-te Nachbar teilt sich mit der Fläche den i-ten und den $i+1$-ten Punktindex.

`Flaeche.ReflektivitaetGueltig` gibt an, ob in der GED-Datei für die Fläche Angaben zur Reflektivität `R` gemacht wurden. Wenn nicht, wird beim Dereferenzieren die dann gültige diffuse Objektfarbe in `Erw.Reflektivitaet` eingesetzt.

Im Gegensatz zum `FlaechenTYP` ist der Typ von Punkten vor und nach dem Dereferenzieren der gleiche. Die meisten Einträge zu `PunktTYP` wurden schon an anderer Stelle des Buches näher erläutert (Abschnitt 18.4.2). Deswegen folgen an dieser Stelle nur die im Text nicht explizit auftauchenden Parameter.

`TexKoordinateLokal` hat den Wert `TRUE`, wenn in der GED-Datei die `TexturKoordinaten` des Punktes definiert waren. Andernfalls erhalten die `TexturKoordinaten` den gleichen Wert wie die `Koordinaten` und `TexKoordinateLokal` ist `FALSE`. Über `EigenschaftsRef` kann nach dem Dereferenzieren der Szene auf die an dieser Stelle gültige Eigenschaft des Teilobjektes zugegriffen werden.

A.9.1.7 Abspeichern von CSG-Ausdrücken: `SolidDefinitionTYP`, `SolidAusdruckTYP` und `SolidTermTYP`

```
SolidDefinitionTYP= RECORD
                     CASE ArtDerSolidDef:ArtDerSolidDefTYP OF
                      Kugel    : (Ku: RECORD
                                        LaengenGradFacet:WORD;
                                        BreitenGradFacet:WORD;
                                    END);
                      Zylinder : (Zy: RECORD
                                        LaengenGradFacet:WORD;
                                        MantelFacet:WORD;
                                        DeckelFacet:WORD;
                                    END);
                      Wuerfel  : (Wu: RECORD Facet:WORD END);
                      Kegel    : (Ke: RECORD
                                        LaengenGradFacet:WORD;
                                        MantelFacet:WORD;
                                        DeckelFacet:WORD;
                                    END);
                      Torus    : (Tor:RECORD
                                        RadiusVerhaeltnis:SINGLE;
                                        LaengenGradFacet:WORD;
                                        BreitenGradFacet:WORD;
                                    END);
                      SolidRef : (re: RECORD
                                       ZeigerZumObjekt: ObjektPtrTYP;
                                    END);
                    END;

SolidAusdruckTYP = RECORD
                     IstBinaer         : BOOLEAN;
                     ErsterInvertiert  : BOOLEAN;
                     ErsterTerm        : SolidTermPtrTYP;
```

```
                    ZweiterInvertiert : BOOLEAN;
                    ZweiterTerm       : SolidTermPtrTYP;
                    Operation         : ArtDerSolidOperationTYP;
                  END;

SolidTermTYP    = RECORD
                   CASE ArtDesSolidTerm:ArtDesSolidTermTYP of
                    SolidDefinition:
                       (SolidDef: SolidDefinitionPtrTYP);
                    CSGAusdruck    :
                       (CSGBaum : SolidAusdruckPtrTYP);
                  END;
```

Die drei Typen dienen zum Abspeichern von CSG-Ausdrücken. Dabei erfolgt die Abspeicherung der Ausdrücke in einem Baum, an dessen Blättern Werte vom Typ `SolidDefinition` „hängen". Je nach `ArtDerSolidDef` können dies vordefinierte Objekte (Kugel, Zylinder ...) oder auch der Aufruf eines Objektes mit Hilfe des Schlüsselworts `SOLID` sein (Abschnitt A.3.4). Die verschiedenen Varianten von `SolidDefinition` sollten selbsterklärend sein.

Abzweigungen in einem CSG-Baum werden mit Hilfe des `SolidAusdruck-TYP` abgespeichert. An den Zweigen hängen immer Variablen vom Typ `SolidTermTYP`, wobei je nach `ArtDesSolidTermTYP` dieser auf Blätter (`SolidDefinitionTYP`) oder CSG-Unterausdrücke (`SolidAusdruckTYP`) zeigt. Am „Hauptzweig", oder besser dem Stamm eines CSG-Baumes, hängt demzufolge auch eine Variable vom Typ `SolidTermTYP`. Auf den Stamm kann dann für ein Teilobjekt mittels `GeometrieDatenTYP.so` zugegriffen werden.

Bleibt noch zu klären, welchen Sinn die Einträge von `SolidAusdruckTYP` machen. `IstBinaer` gibt an, ob der Ausdruck aus einem oder zwei Operanden besteht, `ErsterInvertiert`, ob vor dem ersten Term ein „NICHT" steht. `ErsterTerm` zeigt auf den ersten, `ZweiterTerm` auf den zweiten Term, wobei `ZweiterTerm` nur dann einen sinnvollen Wert hat, wenn `IstBinaer = TRUE` ist. `ZweiterInvertiert` gibt an, ob der zweite Term invertiert ist, wenn er definiert ist. `Operation` schließlich hat, je nachdem ob die die beiden Terme „verUNDet" oder „verODERrt" werden sollen, den Wert `UND_OP` bzw. `ODER_OP`.

A.9.1.8 Abspeichern von Oberflächeneigenschaften: `TexturTYP`, `EigenschaftTYP` und `ErwEigenschaftTYP`

```
TexturTYP       = RECORD CASE Art: ArtDerTexturTYP OF
                    Textur2d, TexturBump: (Name: FileNameTYP);
                    TexturSolid: (SolidArt: ArtDerSolidTexturTYP);
                  END;

EigenschaftTYP  = RECORD
                    Ka,Kd,Ks,Oe,T,N    : SINGLE;
                    OD,OS              : FarbVektorTYP;
                    ArtDerSchattierung: ArtDerSchattierungTYP;
```

```
                Textur          : TexturTYP;
                TexZSkalierung  : SINGLE;
                next            : EigenschaftPtrTYP;
              END;

ErwEigenschaftTYP= RECORD
                Eigensch                    : EigenschaftTYP;
                KaLokal,KdLokal,KsLokal,
                OeLokal,TLokal,NLokal,
                ODLokal,OSLokal,
                ArtDerSchattierungLokal,TexturLokal :BOOLEAN;
                MC_WC,TexMC_WC              : Mat4TYP;
              END;
```

Die Typen TexturTYP und EigenschaftTYP wurden bereits in Abschnitt 18.4.2 erläutert. Da erst nach dem Dereferenzieren der Szene alle Werte von EigenschaftTYP festgelegt sind – sie können ja auch erst durch „Vererbung" festgelegt sein –, wird beim Einlesen einer GED-Datei ein Metatyp ErwEigenschaftTYP verwendet, in dem zusätzlich zu den eigentlichen Eigenschaften diverse Boole'sche Variablen angeben, ob für das entsprechende Teilobjekt die Information explizit definiert ist. Bei der Vererbung von Eigenschaften werden diese Informationen dann nicht überschrieben.

Zusätzlich geben MC_WC und TexMC_WC die Transformationsmatrizen für Koordinaten und Texturen vom Beginn des Objektes bis zu dem Teilobjekt an, für die ErwEigenschaftTYP gültig ist. Nach dem Dereferenzieren sind diese beiden Matrizen nicht mehr nötig, da dann sowohl die Koordinaten als auch die Texturkoordinaten in transformierter Form gespeichert sind. Nach dem Dereferenzieren wird nur noch auf EigenschaftTYP zurückgegriffen, da alle zusätzlichen Informationen von ErwEigenschaftTYP dann nicht mehr benötigt werden.

A.9.1.9 Aufzählungstypen und allgemeine Typen

Die folgenden Aufzählungstypen dienen dazu, die zulässigen Wertebereiche für Parameter der Beleuchtung und Modellierung festzulegen. Weitere Hinweise zu deren Verwendung finden sich in den entsprechenden Kapiteln.

```
ArtDerSchattierungTYP   = (flach,Gouraud,Phong);
ArtDerTexturTYP         = (keine,Textur2d,TexturBump,TexturSolid);
ArtDerSolidTexturTYP    = (Marmor,Holz);
ArtDerSolidOperationTYP = (und_OP,oder_OP);
ArtDerSolidDefTYP       = (Kugel,Zylinder,Wuerfel,Kegel,
                           Torus,SolidRef);
ArtDesSolidTermTYP      = (SolidDefinition,CSGAusdruck);
ArtDerGeometrieDatenTYP = (Solid,PolygonObjekt,ZeichneReferenz);
ArtDerLqTYP             = (AmbienteLq,GerichteteLq,PunktLq,
                           Strahler);
```

Ohne detaillierte Erläuterung folgt auch die Auflistung allgemeiner Typen, die PRIM an den verschiedensten Stellen verwendet.

```
_3TYP          = RECORD CASE INTEGER OF
                   1:  (v: ARRAY[0..2] OF SINGLE);
                   2:  (x, y, z: SINGLE);
                   3:  (R, G, B: SINGLE);
                 END;
_4TYP          = RECORD CASE INTEGER OF
                   1:  (v: ARRAY[0..3] OF SINGLE);
                   2:  (w, x, y, z: SINGLE);
                 END;
Punkt3DTYP     = _3TYP;
Vektor3DTYP    = _3TYP;
FarbVektorTYP  = _3TYP;
Vektor4DTYP    = _4TYP;
Mat4TYP        = ARRAY[0..3,0..3] OF SINGLE;
VierIndizesTYP = ARRAY[0..3] OF WORD;
Vektor2DTYP    = ARRAY[0..1] OF SINGLE;
PolygonTYP     = ARRAY[0..3] OF PunktTYP;
FestKommaTYP   = RECORD CASE BOOLEAN OF
                    TRUE : (Zahl                        : LONGINT);
                    FALSE: (BruchTeil: WORD; GanzTeil : INTEGER);
                  END;
FileNameTYP    = string[80];
```

A.9.1.10 Der GlobaleFlaecheTYP

Zum Abschluß der Darstellung globaler Typen in PRIMGLOB folgt noch die Definition des Typs, mit dessen Hilfe nach dem Dereferenzieren alle Flächen abgespeichert sind.

```
GlobaleFlaecheTYP= RECORD
                     Erw            : ErwFlaechenTYP;
(* Achtung! Erw muss erstes Element des Records sein! *)
                     EigenschaftsRef: EigenschaftPtrTYP;
                     IstRueckflaeche: BOOLEAN;
                     PunktIndizes   : ARRAY[0..3] OF LONGINT;
                     NachbarIndizes : ARRAY[0..3] OF LONGINT;
                     Nachbar1Indizes: ARRAY[0..3] OF LONGINT;
                     FlaechenInhalt : SINGLE;
                     IstElement     : BOOLEAN;
                     IstPatch       : BOOLEAN;
                     PatchIndex     : LONGINT;
                     MaximaleKantenLaenge: SINGLE;
                   END;
```

So gut wie alle Elemente von GlobaleFlaecheTYP wurden bereits an anderer Stelle erläutert (z.B. in den Abschnitten 18.4, 9.3, 18.4.2, usw.). Aus diesem Grund folgt nur eine kurze Zusammenfassung der verschiedenen Elemente. Sie ist in Tabelle A.4 dargestellt.

Es ist wichtig, daß Erw der erste Eintrag von GlobaleFlaecheTYP ist. Auf die ersten beiden Elemente von Erw – Erw.dBA und Erw.FF – wird mit Hilfe

Tabelle A.4 Erläuterung der Datenfelder in GlobaleFlächeTYP

Bezeichnung	Erläuterung
Erw	Werte, die sich GlobaleFlaecheTYP mit FlaecheTYP „teilt“
EigenschaftsRef	Zeiger auf die für diese Fläche gültige Eigenschaft
IstRueckFläche	Flag, das im Laufe der Rendering-Pipeline angibt, ob es sich bei der Fläche um eine Rückfläche handelt
PunktIndizes	Eckpunktindizes der Fläche. Im Gegensatz zu FlaecheTYP, in dem die Punktindizes nur 65536 Werte annehmen können, sind die Punktindizes von globalen Flächen vom Typ LONGINT, da nach dem Dereferenzieren eventuell mehr als 2^{16} Punkte definiert sind.
NachbarIndizes	haben die gleiche Bedeutung wie bei FlaecheTYP. Ebenso wie bei PunktIndizes gilt: Da nach dem Dereferenzieren eventuell mehr als 2^{16} Flächen existieren, ist der Typ der Indizes LONGINT.
Nachbar1Indizes	geben den eventuell zweiten Nachbarn an einer Kante der Fläche an (T-Stücke)
FlaechenInhalt	Flächeninhalt der Fläche
IstElement	gibt nach der Aufteilung in Patches und Elemente an, ob die Fläche ein Element ist
IstPatch	gibt nach der Aufteilung in Patches und Elemente an, ob die Fläche ein Patch ist. Es können alle Kombinationen von IstPatch und IstElement vorkommen.
PatchIndex	gibt bei Flächen, die Element sind, den Index des Patches an, aus denen das Element entstanden ist.
MaximaleKantenLaenge	gibt die maximale Länge der Kante einer Fläche an, die diese nach der Unterteilung haben darf. MaximaleKantenLaenge wird mit Hilfe der Eckpunktkoordinaten und Erw.FacetZahl berechnet.

der Speicherverwaltungsroutinen zugegriffen, ohne die gesamte Fläche zu lesen. Zu diesem Zweck muß aber der Offset relativ zur Fläche gegeben sein. Die entsprechenden Speicherroutinen gehen davon aus, daß dieser Offset für `Erw.dBA = 0` und für `Erw.FF = sizeof(Erw.dBA)` ist (siehe auch Abschnitt 18.4).

A.10 Die Units PRIMREAD und PRIMDERE

Dieses kurze Kapitel beschreibt lediglich zwei Routinen, nämlich `LeseGEDDatei` aus `PRIMREAD` und `DereferenziereSzene` aus `PRIMDERE`. Sie lesen die GED-Dateien ein und erzeugen die internen Datenstrukturen. Es sind jeweils die einzigen Routinen, die in den beiden Modulen im Schnittstellenteil definiert sind.

A.10.1 Die Prozedur LeseGEDDatei

Die Prozedur `LeseGEDDatei` wurde am Anfang dieses Anhangs bereits kurz vorgestellt. Sie übernimmt das Umwandeln der Eingabedatei in die im vorigen Abschnitt beschriebenen Datenstrukturen. Hinter ihr versteckt sich ein kompletter Scanner und Parser, auf den wir allerdings nicht genauer eingehen wollen. Als Literatur zum Thema Compilerbau seien das Buch von Wirth und das der Autorengemeinschaft Klöppel, Paul, Rauch und Ruhland empfohlen [Wir81,KPRR91].

Sollen Änderungen in der Eingabesprache vorgenommen werden, so muß in `PRIMREAD` für die neuen Sprachelemente der Parser entsprechend erweitert werden. Für das restliche Programm ist die Syntax von `GED` allerdings nicht wesentlich. Dort wird ja lediglich auf die erzeugten Datenstrukturen eingegangen. Über den Prozedurkopf von `LeseGEDDatei`

```
PROCEDURE LeseGEDDatei(filename            : STRING;
                       VAR SzenePtr        : ObjektPtrTYP;
                       VAR LichtquellenPtr : LichtquellenPtrTYP;
                       VAR Betrachtung     : BetrachterTYP;
                       VAR WC_NPC, NPC_WC,
                           NPC_DC, WC_DC   : Mat4TYP;
                       VAR Hintergrund     : HintergrundTYP);
```

ist die Schnittstelle zu den restlichen Modulen eindeutig definiert. `SzenePtr` zeigt auf einen Baum von grafischen Objekten und entspricht der kompletten Beschreibung der Szene (siehe Abschnitt A.9.1.2).

Mit Hilfe des `LichtquellenPtr` hat das aufrufende Programm Zugriff auf die in der Eingabedatei definierten Lichtquellen (Abschnitt A.9.1.4). `Betrachtung` beschreibt die Angaben über den Betrachter. Zusätzlich zu `Betrachtung` berechnet `LeseGEDDatei` mit Hilfe der Unit `PRIMVIEW` die entsprechenden Matrizen `WC_NPC`, `NPC_WC`, `NPC_DC`, `WC_DC`, so daß diese direkt verwendet werden können. Die Angaben zum `Hintergrund` sind die letzten, die `LeseGEDDatei` liefert.

A.10.2 Die Prozedur DereferenziereSzene

Die Prozedur DereferenziereSzene generiert aus dem von LeseGEDDatei generierten Baum eine lineare Liste von Flächen und Punkten. Dabei werden alle direkt oder indirekt aus der Szene aufgerufenen polygonalen Teilobjekte in die Listen eingetragen. Nach DereferenziereSzene existiert also zusätzlich zu dem Szenen-Baum ein „Super"-Polygonobjekt, das der Vereinigung aller Polygonobjekte der Eingabedatei entspricht.

Selbst vordefinierte Körper wie die Kugel oder der Kegel sowie komplette CSG-Ausdrücke können in dieses „Super"-Objekt mit aufgenommen werden, wenn sie mit Hilfe von Radiosity- oder z-Buffer-Verfahren dargestellt werden sollen. Ob sie eingetragen werden, bestimmt die Boole'sche Variable SolidsInPolygone, die an DereferenziereSzene übergeben wird. Innerhalb der Unit PRIMDERE existieren Prozeduren, die vordefinierte Körper durch Polygone annähern und diese in die globale Flächenliste eintragen.

Beim Raytracing ist das Darstellen von CSG-Bäumen besonders einfach (siehe Teil 4). Dies war die Hauptmotivation, solche Strukturen auch in die Eingabesprache mit aufzunehmen.

Das Darstellen von CSG-Ausdrücken auf Polygonbasis ist allerdings ungleich komplizierter und erfordert einen immensen Aufwand, der den Umfang dieses Buches bei weitem überschritten hätte. Aus diesem Grund haben wir ein sehr einfaches Verfahren realisiert, das allerdings nur in speziellen Fällen „gute" Ergebnisse liefert:

1. Die ODER-Verknüpfung von zwei Objekten bedeutet nichts anderes, als beide Objekte darzustellen.
2. Das „NICHT" vor einem CSG-Element bedeutet, daß man nicht die Vorder- sondern die Rückflächen des Elementes sieht.
3. „UND"-Ausdrücke sind „kompliziert" und werden nicht unterstützt.

Hierzu drei Beispiele: `(Kegel ODER Kugel)` wird korrekt dargestellt, `(NICHT Kugel)` stellt die Rückfläche einer Kugel dar und `(Kegel ODER (Kugel UND Wuerfel)` stellt nur den Kegel dar, der UND-Term wird ignoriert.

Es ist auf jeden Fall sinnvoll, auch beim polygonalen Rendern CSG-Ausdrücke korrekt dazustellen. Hier tut sich für Sie ein weites Betätigungsfeld auf. Es ist aber sichergestellt, daß nur in der Unit PRIMDERE Änderungen vorgenommen werden müssen.

Der Prozedurkopf von DereferenziereSzene ist als

```
PROCEDURE DereferenziereSzene(
              SzenePtr            : ObjektPtrTYP;
              SolidsInPolygone    : BOOLEAN;
              VAR AnzahlFlaechen  : LONGINT;
              VAR AnzahlPunkte    : LONGINT;
              VAR EigenschaftListe: EigenschaftPtrTYP);
```

definiert. Als Eingabeparameter dienen SzenePtr, also ein Zeiger auf die eingelesene Szene und die bereits angesprochene Variable SolidsInPolygone.

Die eigentlichen Ergebnisse des Dereferenzierens, also die globale Flächen- und Punktliste, tauchen nicht direkt in der Parameterliste von `DereferenziereSzene` auf. Vielmehr gibt die Prozedur nur die Anzahl der Punkte und die Anzahl der Flächen zurück. Auf die beiden Listen kann dann im weiteren Verlauf des Programms mit Hilfe der Prozeduren `PunktAnGlobaleListe`, `PunktAusGlobalerListe`, `PunktInGlobaleListe`, `FlaecheAnGlobaleListe`, `FlaecheAusGlobalerListe` und `FlaecheInGlobaleListe`, die in `PRIMMEM` definiert sind, zugegriffen werden (Anhang C). Die globalen Listen stehen nicht im „normalen" PC-Speicher, sondern im XMS-Speicher.

Im Gegensatz zu Punkt- und Flächenliste steht die `EigenschaftListe`, als „normale" verkettete Liste zur Verfügung. Diese Liste besteht aus allen verschiedenen Eigenschaften, auf welche die Flächen zurückgreifen. Der Typ `EigenschaftTYP` ist weiter vorne in diesem Kapitel genauer beschrieben.

A.11 Format der Texturdateien

Es gibt diverse Möglichkeiten, Bilder, und um nichts anderes handelt es sich bei 2D-Texturen, abzuspeichern. So existieren mehrere Standards, die genau beschreiben, in welchem Format die Bildinformation abgespeichert werden soll. Zu diesen Standards gehören z.B. das Windows „.BMP"-Format, das Targa-Format („.TGA"), das Tiff-Format („.TIF") etc.

All diese Formate sind mehr oder weniger universell und daher relativ mühsam korrekt zu lesen, wenn man wirklich alle möglichen Unterformate eines bestimmten Formates einlesen können will. So existieren z.B. beim Tiff-Format diverse verschiedene Kompressionsarten, für die jeweils in der Einleseroutine ein Dekomprimierer implementiert werden müßte.

Um einerseits ein universelles Format zu benutzen, andererseits den Aufwand für das Einlesen von Texturen nicht ins Uferlose wachsen zu lassen, haben wir uns auf unkomprimierte 4- und 8-Bit Windows BitMap-Dateien beschränkt. Sämtliche, mit Microsoft Windows mitgelieferten BMP-Dateien sind in diesem Format gespeichert. Mit entsprechenden „Windows-Mal-Programmen" können so auf einfache Art neue Texturen für `PRIM` erzeugt werden.

Wie das genannte Format genau definiert ist, kann der entsprechenden Literatur entnommen werden [Mic]. Im Prinzip bestehen solche Dateien aus einem Header, einer Farbtabelle und einem Bild, für das pro Pixel ein halbes bzw. ein Byte gespeichert ist. Die für jedes Pixel gespeicherte Farbe einer Textur wird auch als „Texel" (texture element) bezeichnet.

BMP-Dateien, die nur ein halbes Byte pro Pixel belegen, werden direkt beim Lesen auf ein Byte pro Pixel „aufgeblasen", so daß sich 8-Bit-Texturen und 4-Bit-Texturen nach dem Lesen nicht mehr unterscheiden. Dies ist zwar Speicherplatzverschwendung, macht das Lesen eines Texels beim eigentlichen Texturieren eines Pixels aber einfacher.

Da Texturen unabhängig von der Darstellung durch `PRIM` generiert werden können, ist es auch möglich, eine optimale Farbtabelle zu berechnen. Deswegen

können trotz der 8 oder sogar nur 4 Bit ausreichend „bunte“ Texturen verwendet werden. Ein nicht unwesentliches Argument für diese Farbindex-Texturen war übrigens, daß sie nur etwa ein Drittel des Diskettenplatzes von Echtfarb-Texturen benötigen – ein Umstand, den die Binderei zu schätzen wußte, die sich schlicht geweigert hätte, mehr als eine Diskette zu diesem Buch zu legen.

Wie `PRIM` die Texturen intern weiter bearbeitet, ist in Abschnitt C.2.5 näher erläutert.

B. Syntax der Grafischen Eingabesprache

Dieser Anhang enthält die genaue Definition der Syntax der Grafischen Eingabesprache GED. Die Semantik der einzelnen Befehle wurde bereits in Anhang A beschrieben.

B.1 Allgemeine Festlegungen

1. Es wird nicht zwischen Groß- und Kleinschreibung unterschieden. Zur besseren Kennzeichnung sind Schlüsselworte im Text allerdings meist groß geschrieben.
2. Trennzeichen zwischen Worten sind das Leerzeichen „␣“, das Tabulatorzeichen und das Zeichen für die neue Zeile.
3. Kommentare beginnen mit „//“ und enden am Zeilenende.

B.2 Die Syntax der Beschreibung

Die Beschreibung wird in einer der Backus-Naur-Form ähnlichen Notation dargestellt. Dazu werden die folgenden Metasymbole verwendet:

{ } | ( ) ::= [ ] ..

Ist A ein zulässiger Ausdruck, dann bedeuten die verschiedenen Klammerungen:

{ A }	A kommt beliebig häufig wiederholt (auch kein Mal) vor
{ A }n	A kommt genau n Mal vor
[A]	A kommt kein oder ein Mal vor
A | B	A oder B

Die runden Klammern () dienen zur Vorrangregelung. ::= entspricht einer Definition der Beschreibungssyntax. Fett gedruckte Symbole (z.B. **Zeichen**) sind reservierte Worte bzw. Zeichen der Sprache.

a..z beschreibt alle Zeichen von a bis z. Bezeichner mit Kleinbuchstaben werden weiter aufgelöst.

B.3 Sprachdefinition

Grafik-Eingabe-Datei	::=	Version Rumpf .
Version	::=	**GED_VERSION** GanzeZahl ;
Rumpf	::=	Objektliste Betrachter Lichtquellen Szene
GanzeZahl	::=	Ziffer { Ziffer }
Objektliste	::=	{ Objekt }
Lichtquellen	::=	[AmbienteLq | GerichteteLq | PunktLq | Strahler] { Lichtquellen }
Szene	::=	**SZENE** Identifier ; [Hintergrund] { Teilobjekt } **END**
Ziffer	::=	**0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
Objekt	::=	**OBJEKT** Identifier ; { Teilobjekt } **ENDOBJEKT** ;
Betrachter	::=	[((**AUGE** 3Vektor) | (**REFERENZPUNKT** 3Vektor) | (**UPVEKTOR** 3Vektor) | (**BILDABSTAND** ReelleZahl) | (**VORDERCLIPEBENE** ReelleZahl) | (**HINTERCLIPEBENE** ReelleZahl) | (**BILDBREITE** ReelleZahl) | (**BILDSCHIRMFENSTER** GanzeZahl {, GanzeZahl }3) | (**PIXELRATIO** ReelleZahl)) ;] { Betrachter }
AmbienteLq	::=	**AMBIENTELQ** FarbVektor ;
GerichteteLq	::=	**GERICHTETELQ** FarbVektor 3Vektor ;
PunktLq	::=	**PUNKTLQ** FarbVektor 3Vektor 2Vektor ;
Strahler	::=	**STRAHLER** FarbVektor 3Vektor 3Vektor 2Vektor [ReelleZahl [ReelleZahl]] ;
Identifier	::=	ErstesZeichen { Zeichen }
Hintergrund	::=	{ Eigenschaft } [**HINTERGRUND** [Farbvektor, Farbvektor] ;]
Teilobjekt	::=	{ Eigenschaft } [Geometriedaten]
3Vektor	::=	ReelleZahl {, ReelleZahl }2
ReelleZahl	::=	[+] | [−] GanzeZahl [. GanzeZahl] [**E** [+] | [−] GanzeZahl]
FarbVektor	::=	3Vektor
2Vektor	::=	ReelleZahl , ReelleZahl

```
ErstesZeichen   ::= a..z | A..Z | _
Zeichen         ::= a..z | A..Z | _ | Ziffer | \ | :
Eigenschaft     ::= (   (KA | KD | KS | OE | T | N) ReelleZahl
                      | ((OS | OD) FarbVektor)
                      | SCHATTIERUNG
                        (FLACH | GOURAUD | PHONG)
                      | Transformation
                      | Textur
                    )  ;
Geometriedaten  ::= Solid | Polygonobjekt | ZEICHNE Identifier ;
Transformation  ::= PushPop | Translation | Rotation | Skalierung
PushPop         ::= (PUSH | POP)
                    { TEXTURKOORDINATEN
                      | KOORDINATEN
                      | EIGENSCHAFTEN
                      { , TEXTURKOORDINATEN
                        | KOORDINATEN
                        | EIGENSCHAFTEN  } }
Translation     ::= VERSCHIEBE [[MIT] TEXTUR] 3Vektor
Rotation        ::= DREHE [[MIT] TEXTUR] (X | Y | Z)
                    , ReelleZahl
Skalierung      ::= SKALIERE [[MIT] TEXTUR]
                    (ReelleZahl | 3Vektor)
Textur          ::= NOMAP
                    | (MAP | BUMPMAP) filename
                    | SOLIDMAP (MARMOR | HOLZ)
Solid           ::= SolidTerm ;
Polygonobjekt   ::= PunktListe Flächenliste
SolidTerm       ::= (SolidDefinition | CSGAusdruck)
SolidDefinition ::= KUGEL [FACET GanzeZahl [,  GanzeZahl]]
                    | ZYLINDER [FACET GanzeZahl
                      [,  GanzeZahl [,  GanzeZahl]]]
                    | WUERFEL [FACET GanzeZahl]
                    | KEGEL [FACET GanzeZahl
                      [,  GanzeZahl [,  GanzeZahl]]]
                    | TORUS ReelleZahl
                      [FACET GanzeZahl [,  GanzeZahl]]
                    | SOLID Identifier
```

CSGAusdruck	::=	([**NICHT**] Solidterm
		[(**UND** | **ODER**)[**NICHT**] Solidterm])
PunktListe	::=	**PUNKTE** Punkt Punkt Punkt { Punkt }
Flächenliste	::=	**FLAECHEN** Fläche { Fläche }
Punkt	::=	3Vektor [2Vektor | 3Vektor] [**SCHARF**] ;
Fläche	::=	Punktindizes [**FLIP**] [**FACET** GanzeZahl]
		[**R** 3Vektor] [**B** 3Vektor] ;
Punktindizes	::=	GanzeZahl , GanzeZahl ,
		GanzeZahl { , GanzeZahl}

B.4 Liste der reservierten Worte

Die folgende Liste enthält die reservierten Worte der Grafischen Eingabesprache GED.

```
AMBIENTELQ
AUGE
B
BILDABSTAND
BILDBREITE
BILDSCHIRMFENSTER
BUMPMAP
DREHE
EIGENSCHAFTEN
END
ENDOBJEKT
FACET
FLACH
FLAECHEN
FLIP
GED_VERSION
GERICHTETELQ
GOURAUD
HINTERCLIPEBENE
HINTERGRUND
HOLZ
KA
KD
KEGEL
KOORDINATEN
KS
KUGEL
MAP
MARMOR
MIT
N
NICHT
NOMAP
OBJEKT
OD
ODER
OE
OS
PHONG
PIXELRATIO
POP
PUNKTE
PUNKTLQ
PUSH
R
REFERENZPUNKT
SCHARF
SCHATTIERUNG
SKALIERE
SOLID
SOLIDMAP
STRAHLER
SZENE
T
TEXTUR
TEXTURKOORDINATEN
TORUS
UND
UPVEKTOR
VERSCHIEBE
VORDERCLIPEBENE
WUERFEL
X
Y
Z
ZEICHNE
ZYLINDER
```

C. Hardwareabhängige Module von PRIM

Das größte Problem bei der Implementierung von Rendering-Algorithmen mit Hilfe des z-Buffers ist der nicht unerhebliche Speicherverbrauch. Einerseits benötigen etwas aufwendigere Objekte, abhängig von der Anzahl der Polygone, relativ viel Speicher, andererseits ist der z-Buffer-Algorithmus an sich speicherhungrig. Pro Pixel wird die Tiefeninformation gespeichert. Bei einer Auflösung von 800×600 Pixel und einer 16-Bit-Zahl pro z-Buffer-Eintrag kommt man allein für den z-Buffer auf 937.5 KByte Speicherverbrauch.

Für den heute üblichen Speicherausbau eines PC stellt die Menge an sich nicht das Problem dar. Vielmehr ist es das Betriebssystem DOS, das dem geplagten Programmierer das Leben schwer macht. Oberhalb der magischen, oder passender: der historischen 640-KByte-Grenze wird's problematisch. Abhilfe schaffen hier spezielle Treiber, die es unter DOS ermöglichen, auf den gesamten installierten Speicher zuzugreifen.

Wir haben für unsere Implementierung den „XMS"-Treiber von Microsoft verwendet, der z.B. mit jedem MS-Windows ausgeliefert wird. Dieser Treiber stellt spezielle Routinen zur Verfügung, mit deren Hilfe es möglich ist, Speicher in, im und aus dem Bereich oberhalb der 640 KByte zu kopieren. Hierbei wird noch zwischen dem Bereich von 640 KByte bis 1 MByte und dem Bereich ab 1 MByte unterschieden. Wir verwenden ausschließlich den zweiten Bereich.

Vor der Beschreibung der XMS-Speicher-Verwendung folgt allerdings ein Abschnitt, der sich an alle wendet, die

- einen PC mit weniger als ca. 4 MByte Speicher besitzen oder
- eine Portierung der Software auf eine andere Hardware oder einen anderen Compiler vornehmen wollen.

Zur ersten Gruppe: Alle Routinen, die mehr als den üblichen PC-Speicher verwenden, sind ausschließlich im Modul `PRIMMEM` realisiert. Dabei greifen die diversen Routinen lediglich auf eine einzige Funktion `CallXMS` zu, welche die eigentliche Speicherverwaltung übernimmt. Ersetzt man diese Funktion durch eine eigene Funktion, die z.B. auf der Festplatte eine Datei als „Speichererweiterung" anlegt, so ist es auch mit „normalem" Speicherausbau möglich, den Rest der Software zum Laufen zu bringen. Was die Funktion `CallXMS` im einzelnen macht, wird weiter unten beschrieben.

Zur zweiten Gruppe: Wenn Sie einer der glücklichen Computerbenutzer sind, die den gesamten Speicher ganz einfach als Feld ansprechen können –

sogar für PC's gibt es Compiler, mit denen dies möglich ist – so sollten alle Routinen, die eigentlich auf ein großes Feld zurückgreifen, von Ihnen umgeschrieben werden. Dies sind insbesondere die Routinen, die den z-Buffer und den item-Buffer verwenden.

C.1 CallXMS, die Funktion zum Ansprechen des erweiterten Speichers

Im Interfaceteil von PRIMMEM sind eine relativ große Anzahl von Prozeduren realisiert, die letzten Endes alle auf die Funktion CallXMS zurückgreifen. Diese Funktion stellt die eigentliche Verbindung zum XMS-Speicher her. Wegen ihrer Kürze folgt ihr gesamtes Listing:

```
FUNCTION CallXMS: BOOLEAN;
BEGIN
  ASM
    mov   ax,XMS_ax; mov   bx,XMS_bx;
    mov   dx,XMS_dx; mov   ds,XMS_ds;
    mov   si,XMS_si; push  bp;
    call  xmsProc;   pop   bp;
    mov   XMS_ax,ax; mov   XMS_dx,dx;
  END;
  CallXMS:=XMS_ax=1;
END;
```

Wie man sieht, dienen die in PRIMMEM definierten globalen Variablen XMS_ax, XMS_bx, XMS_dx und XMS_ds zur Parameterübergabe. Diese werden in die entsprechenden Register des Prozessors kopiert und anschließend die Routine xmsProc aufgerufen. xmsProc ist ebenfalls eine Variable vom Typ PROCEDURE. Wie xmsProc mit einem Wert belegt wird, folgt weiter unten.

Die Routine liefert zwei Ergebnisse, die in XMS_ax und XMS_dx kopiert werden. Der Wert der Funktion CallXMS ist TRUE, wenn der Aufruf von xmsProc erfolgreich war, im anderen Fall ist er FALSE.

Laut Microsoft XMS-Spezifikation gelangt man an die Einsprungadresse des XMS-Treibers mit Hilfe folgender Prozedur:

```
PROCEDURE InitXMS;
VAR reg: registers;
BEGIN
  reg.ax:=$4300; intr($2F,reg);
  IF reg.al<>$80 THEN Fehler('XMS-Treiber nicht geladen. ');
  reg.ax:=$4310; intr($2F,reg);
  IF reg.ax=0 THEN Fehler('XMS-Einsprung nicht ermittelbar.');
  @XMSProc:=POINTER(LONGINT(reg.es) shl 16 +reg.bx);
END;
```

Über den Softwareinterrupt $2F wird also festgestellt, ob der XMS-Treiber geladen ist und anschließend die Einsprungadresse ermittelt. InitXMS wird bei

Tabelle C.1 Registerbelegung beim Aufruf der XMS-Speicherverwaltungsroutinen

Funktion	Eingabeparameter	Ergebnis
Speicher allokieren	XMS_ax=$900	–
	XMS_dx=	Handle des
	Anzahl KBytes	Speichers
Speicher freigeben	XMS_ax=$A00	–
	XMS_dx=Handle des	–
	Speichers	
Größe von allokiertem	XMS_ax = $F00	–
Speicher verändern	XMS_bx = neue Größe in	–
	KBytes	
	XMS_dx = Handle des	–
	Speichers	
Speicher kopieren	XMS_ax = $B00	–
	XMS_ds:XMS_si =	veränderter
	XMS_Parameterblock	Parameterblock

der Initialisierung der Unit automatisch aufgerufen. Ist mit Hilfe der Datei `CONFIG.SYS` kein XMS-Treiber eingebunden, so ergibt sich eine Fehlermeldung und das Programm bricht ab.

Anschließend kann man mit Hilfe von `CallXMS` alle Funktionen des XMS-Treibers benutzen, vorausgesetzt, in den Variablen `XMS_..` stehen die entsprechenden Werte. Von der Vielzahl der möglichen Funktionen haben wir in unserer Implementierung lediglich vier benutzt:

1. XMS-Speicher anfordern,
2. XMS-Speicher freigeben,
3. bereits angeforderten XMS-Speicher in der Größe verändern und
4. Speicher kopieren.

Der Zugriff auf den XMS-Speicher erfolgt über sogenannte „Handles", deren Funktion ähnlich der von Dateivariablen ist. Auf einen angeforderten Speicherblock hat man nur mit Hilfe des „Handles" Einfluß. Die Größe des Speichers, den man mit Hilfe des XMS-Treibers anfordern kann, muß ein Vielfaches von einem KByte sein, kleinere „Brocken" sind nicht möglich. Die Tabelle C.1 gibt einen Überblick über die Registerwerte bei den vier verschiedenen Aufrufen. Bis auf die Funktion „Speicher kopieren", die weiter unten genauer dargestellt wird, sollte diese Tabelle als Erläuterung ausreichen.

Kurz vor der Fertigstellung des Buches tauchte ein Problem auf, das an dieser Stelle nicht verschwiegen werden soll. Bei der Entwicklung der Software haben wir einen XMS-Treiber mit der Versionsnummer 2.77 verwendet. Mit dieser Version von `HIMEM.SYS` arbeitet die Funktion $F000 (Größe von belegtem Speicher verändern) einwandfrei. Es stellte sich dann aber heraus, daß mit

dem Treiber in der Version 3.07, der mit Windows 3.1 ausgeliefert wird, diese Funktion nicht mehr einwandfrei funktioniert.

Um trotzdem mit dieser Version arbeiten zu können, wird in PRIMMEM die entsprechende XMS-Funktion nicht benutzt. Vielmehr geht die Prozedur ErweitereXMSSpeicher wie folgt vor:

1. Speicherbereich neuer Größe allokieren
2. alten Speicherinhalt in neuen Speicherbereich kopieren
3. alten Speicherbereich freigeben

Dieses Verfahren hat natürlich den Nachteil, daß *während* der Erweiterung sowohl Speicher der alten wie auch Speicher der neuen Größe belegt ist. Unter Umständen führt dies dazu, daß der verfügbare Speicher nicht ausreicht.

„Speicher kopieren" (XMS_ax=$B00) verwendet zur Parameterübergabe in XMS_ds:XMS_si einen Zeiger auf die in PRIMMEM definierte Struktur XMS_Parameterblock. In ihr sind die für das Speicherkopieren nötigen Informationen enthalten.

```
TYPE XMS_ParameterBlockTYP = RECORD
                               BlockLaengeInByte: LONGINT;
                               HandleQuelle     : WORD;
                               OffsetQuelle     : LONGINT;
                               HandleZiel       : WORD;
                               OffsetZiel       : LONGINT;
                             END;
```

Mit Hilfe von BlockLaengeInByte wird angegeben, wieviele Bytes übertragen werden sollen. Dabei muß BlockLaengeInByte *gerade* sein. Da BlockLaengeInByte vom Typ LONGINT ist, können auch sehr große Speicherbereiche kopiert werden.

HandleQuelle gibt das „Handle" des Quellspeicherbereiches an. Ab welchem Byte des Quellbereiches kopiert werden soll, geht aus OffsetQuelle hervor. HandleZiel und OffsetZiel sind die entsprechenden Angaben für den Zielbereich. Im Normalfall geben OffsetQuelle und OffsetZiel den Offset der Bereiche in Bytes an. Wenn das zugehörige Handle 0 ist, wird Offset als normaler Segment:Offset-Zeiger im PC-Speicher gedeutet. In unserer Implementierung ist immer einer der beiden Offsets Null, d.h. es wird immer aus bzw. in die ersten 640 KByte transferiert.

Ein Beispiel: Die Bytes 200 bis 299 aus dem XMS-Bereich mit dem Handle $0815 sollen in die Variable a, die sich im „normalen" PC-Speicher befindet, kopiert werden. Der entprechende Aufruf von CallXMS hätte dann die Form

```
XMS_ParameterBlock.BlockLaengeInByte:=100;
XMS_ParameterBlock.HandleQuelle     :=$0815;
XMS_ParameterBlock.OffsetQuelle     :=200;
XMS_ParameterBlock.HandleZiel       :=0;
XMS_ParameterBlock.OffsetZiel       :=LONGINT(@a);
XMS_AX:=$B00;
IF NOT CallXMS THEN Fehler('Fehler beim Lesen aus XMS');
```

Die Variablen XMS_ds und XMS_si werden im Initialisierungsteil von PRIMMEM auf Segment und Offset von XMS_ParameterBlock gesetzt.

Nachdem nun die wesentlichen in PRIMMEM verwendeten XMS-Routinen erläutert sind, folgt im nächsten Abschnitt eine Auflistung der Routinen, die mit Hilfe des XMS-Speichers arbeiten und im Interfaceteil von PRIMMEM definiert sind.

C.2 Die Routinen von PRIMMEM

Die Routinen von PRIMMEM können in fünf Unterklassen aufgeteilt werden:

- Routinen zum Ansprechen von z-Buffer, item-Buffer und Deltaformfaktor-Buffer,
- Routinen zum Laden und Lesen von Texturen,
- Routinen zum Ansprechen von Speicher für polygonale Teilobjekte,
- Routinen zum Ansprechen von globaler Punkt- und Flächenliste und
- Routinen zum gezielten Ansprechen von FF und dBA einer globalen Fläche.

Außer diesen Routinen sind im Interfaceteil von PRIMMEM noch die beiden Variablen AnzahlDefPunkte und AnzahlDefFlaechen sowie die Konstante MaximaleZIBufferBreite definiert. Die beiden Variablen geben die augenblickliche Anzahl der Punkte und Flächen in den entsprechenden globalen Listen an. Sie werden in den Routinen zum Ansprechen von globaler Punkt- und Flächenliste verändert. MaximaleZIBufferBreite sollte bei Routinen, die den z-Buffer oder den item-Buffer benutzen, nicht überschritten werden. Notfalls muß die Konstante verändert werden. Ihr Wert beträgt zur Zeit 1024.

C.2.1 Routinen zum Ansprechen des z-Buffers

Auf den z-Buffer kann pixel- oder spanweise zugegriffen werden. Vor dem ersten Zugriff muß der z-Buffer mit Hilfe von AllokiereZBuffer angefordert werden. Im Programm werden verschiedene z-Buffer verwendet: Einerseits im Radiosity-Teil bei dem Rendern auf die Singleplane, andererseits bei der Darstellung selbst. Nach erfolgter Radiosiy-Iteration muß in diesem Fall der z-Buffer erst mit GebeZBufferFrei wieder freigegeben und dann für das eigentliche Rendern erneut, i.a. mit veränderter Größe, angefordert werden.

Für den z-Buffer sind die Prozeduren AllokiereZBuffer, LoescheGesamtenZBuffer, GebeZBufferFrei, LeseZBufferZeile, SchreibeZBufferZeile und ZBufferTest zuständig. Es folgt jeweils eine detaillierte Beschreibung der Parameter und der Funktionalität in tabellarischer Form.

Syntax:

```
PROCEDURE AllokiereZBuffer(Breite, Hoehe, VerschiebungX,
    VerschiebungY: WORD);
```

Parameter:

`Breite, Hoehe:`	Breite und Höhe des z-Buffers in „Pixeln“
`VerschiebungX, VerschiebungY:`	Koordinate des linken oberen „Pixels“

Funktionalität:

`AllokiereZBuffer` reserviert einen Speicherbereich von `Breite` × `Hoehe` Integer-Zahlen, auf den anschließend mit Hilfe der Prozeduren `LeseZBufferZeile`, `SchreibeZBufferZeile`, `LeseZIBufferZeile`, `SchreibeZIBufferZeile` sowie der Funktion `zBufferTest` zugegriffen werden kann.

Um in diesen Funktionen mit absoluten Bildschirmkoordinaten arbeiten zu können, aber trotzdem nur Speicher für das tatsächlich verwendete Fenster zu reservieren, werden die Variablen `VerschiebungX` und `VerschiebungY` verwendet. Mit ihrer Hilfe erfolgt die Umrechnung der absoluten Bildschirmkoordinaten in die relativen Fensterkoordinaten.

Syntax:

```
PROCEDURE LoescheGesamtenZBuffer;
```

Funktionalität:

Nach der Anforderung mit `AllokiereZBuffer` kann mit Hilfe dieser Prozedur der gesamte z-Buffer auf den Wert $FFFF („weit weg“) initialisiert werden.

Syntax:

```
PROCEDURE GebeZBufferFrei;
```

Funktionalität:

Bevor ein z-Buffer mit veränderter Größe angefordert werden soll, muß mit Hilfe dieser Routine der alte z-Buffer freigegeben werden.

Syntax:

```
PROCEDURE LeseZBufferZeile(x1, x2, y: INTEGER; VAR buf);
```

Parameter:

`x1, x2:`	x-Koordinate des ersten und letzten Pixels der Zeile, die gelesen werden soll
`y:`	y-Koordinate der zu lesenden Zeile
`buf:`	Buffer, in den die gelesene Zeile geschrieben wird. `buf` sollte vom Typ `ARRAY[0..maxX-1] OF INTEGER` definiert sein, wobei `maxX` ≥ `x2` sein muß.

Funktionalität:

Die z-Bufferwerte in Höhe `y` werden von `x1` bis `x2` gelesen und in `buf` geschrieben. Die Werte von `buf[0..x1-1]` und von `buf[x2+1..Bufferende]` bleiben unverändert.

Syntax:
`PROCEDURE SchreibeZBufferZeile(x1, x2, y: INTEGER; VAR buf);`

Parameter:
Die Parameter sind mit denen von `LeseZBufferZeile` identisch.

Funktionalität:
Diese Prozedur ist das Gegenstück zu `LeseZBufferZeile`. Anstatt zu Lesen werden die z-Werte aus `buf` in den z-Buffer geschrieben.

Syntax:
`FUNCTION ZBufferTest(x, y: INTEGER; z: WORD): BOOLEAN;`

Parameter:

`x, y, z:`	x-, y- und z-Komponente des zu testenden Pixels
`Ergebnis:`	`TRUE`, wenn das Pixel zu sehen ist, `FALSE`, wenn es verdeckt ist

Funktionalität:
`ZBufferTest` testet, ob ein Pixel zu sehen ist oder nicht. Wenn es zu sehen ist, wird der z-Buffer an der Stelle (x, y) auf z gesetzt, ansonsten bleibt er unverändert.

C.2.2 Routinen zum Ansprechen des item-Buffers

Die Routinen zum Ansprechen des item-Buffers sind denen zum Ansprechen des z-Buffers sehr ähnlich. Im einzelnen sind es:

Syntax:
`PROCEDURE AllokiereIBuffer(PixelProKante: WORD);`

Parameter:

`PixelProKante:`	Auflösung des item-Buffers

Funktionalität:
Die Routine entspricht der Prozedur `AllokiereZBuffer` für den item-Buffer. Im Gegensatz zu dieser wird allerdings ausschließlich die Kantenlänge des item-Buffers übergeben, da der item-Buffer (also die „Singleplane“) immer quadratisch ist und bei $(0, 0)$ beginnt.

Syntax:
`PROCEDURE LoescheGesamtenIBuffer;`

Funktionalität:
Alle Werte des item-Buffer werden auf -1 gesetzt. Bei der Formfaktorberechnung deutet eine -1 in der Singleplane an, daß dort kein Polygon zu „sehen“ ist.

Syntax:
`PROCEDURE GebeIBufferFrei;`

Funktionalität:
Äquivalent zu `GebeZBufferFrei`.

Syntax:
`PROCEDURE LeseIBufferZeile(y: INTEGER; VAR IZeile);`

Parameter:

`y:`	y-Koordinate der Zeile, die aus dem item-Buffer gelesen werden soll
`IZeile:`	Variable, in welche die Zeile geschrieben wird. `IZeile` muß als `ARRAY[0..maxX-1] OF LONGINT` definiert sein, wobei `maxX` $\geq$ die aktuelle Singleplane-Auflösung sein muß.

Funktionalität:
Liest eine komplette Zeile aus dem item-Buffer. Siehe auch `LeseZBufferZeile`.

Syntax:
`PROCEDURE ZeigeIBuffer;`

Funktionalität:
`ZeigeIBuffer` stellt den aktuellen item-Buffer (die „Singleplane“) in der linken oberen Ecke des Bildschirms dar. Dabei erhalten die Pixel eine Farbe, die aus dem item-Bufferwert hervorgeht.

C.2.3 Routinen zum gemeinsamen Ansprechen von z- und item-Buffer

Bei der Radiosity-Iteration werden meistens der z-Buffer und der item-Buffer gemeinsam verändert. Um unnötige Aufrufe zu sparen, wurden zwei Routinen implementiert, die beide Buffer gemeinsam behandeln.

Syntax:
`PROCEDURE LeseZIBufferZeile(x1, x2, y: INTEGER; VAR zbuf, ibuf);`

Parameter:
Die Parameter sind im wesentlichen mit denen von `LeseZBufferZeile` identisch. Es werden zwei Buffer, einer für die z-Werte, der andere für die item-Werte übergeben.

Funktionalität:
Lesen einer Zeile von `x1` bis `x2` in Höhe `y` aus z- und item-Buffer.

Syntax:
```
PROCEDURE SchreibeZIBufferZeile(x1, x2, y: INTEGER;
    VAR zbuf, ibuf);
```

Parameter:
Parameter wie `LeseZIBufferZeile`.

Funktionalität:
Schreiben einer Zeile von `x1` bis `x2` in Höhe `y` in z- und item-Buffer. Gegenstück zu `LeseZIBufferZeile`.

C.2.4 Routinen zum Ansprechen des Deltaformfaktor-Buffers

Da sich die Deltaformfaktoren während der Radiosity-Iteration nicht ändern, werden sie ebenfalls im XMS-Speicher gehalten. Für sie muß kein gesonderter Speicherbereich reserviert werden. Dies geschieht beim Reservieren des item-Buffers automatisch. Die Anzahl der zu speichernden Faktoren geht immer aus der Auflösung des item-Buffers hervor. Die Routinen zum Ansprechen des Deltaformfaktor-Buffers sind:

Syntax:
`PROCEDURE LesedFFBufferZeile(y: INTEGER; VAR dFFZeile);`

Parameter:

`y`	y-Koordinate der Zeile, die gelesen werden soll
`dFFZeile`	Gelesene Zeile

Funktionalität:
`LesedFFBufferZeile` liest eine gesamte Zeile der Deltaformfaktoren in `dFFZeile`. Da nur die obere Hälfte der Deltaformfaktoren gespeichert ist, muß y entsprechend korrigiert werden, wenn es größer als die aktuelle Auflösung der Singleplane ist. Sei sp_{size} die Auflösung der Singleplane und y_{voll} die echte y-Koordinate. Dann ergibt sich `y` zu

$$y = \begin{cases} y_{voll} & \text{für } y < sp_{size} \text{ DIV } 2 \\ sp_{size} - y_{voll} - 1 & \text{sonst} \end{cases}$$

Da die Deltaformfaktoren für die grobe und feine Singleplane hintereinander abgespeichert vorliegen, muß zu `y` noch $(sp_{size} + 1)$ `DIV 2` addiert werden, wenn die Deltaformfaktoren der feinen Singleplane gelesen werden sollen. Siehe auch `BerechneFormfaktoren` der Unit `PRIMRAD`.

Syntax:
`PROCEDURE SchreibedFFBufferZeile(y: INTEGER; VAR dFFZeile);`

Parameter:
Die Parameter sind identisch mit denen von `LesedFFBufferZeile`.

Funktionalität:
`SchreibedFFBufferZeile` dient zum Initialisieren der Deltaformfaktoren. Die Routine ist das Gegenstück zu `LesedFFBufferZeile`. Siehe auch `BerechnedFFs` der Unit `PRIMRAD`.

C.2.5 Routinen zum Ansprechen von Texturen

In Anhang A wurde bereits beschrieben, welches Format Texturen haben müssen, um in `PRIM` benutzt werden zu können. Um mit diesen Texturen zu arbeiten, wurden die folgenden Prozeduren definiert:

```
PROCEDURE LadeTextur(VAR name: FileNameTYP);
PROCEDURE LeseTexturPixel(x, y: INTEGER; VAR r, g, b: BYTE);
PROCEDURE LeseTexturWert(x, y: INTEGER; VAR h: BYTE);
```

Mit `LadeTextur` wird eine Textur von der Platte in den XMS-Speicher geladen. Die beiden anderen Prozeduren greifen immer auf die zuletzt geladene Textur zu. Wenn die Textur bereits mittels eines vorigen Aufrufs geladen wurde, so wird nicht mehr auf die Platte zugegriffen. In diesem Fall erfolgt nur eine Aktualisierung von internen Variablen, so daß nach dem Aufruf von `LadeTextur` die entsprechende „alte" Textur von `LeseTexturPixel` und `LeseTexturWert` benutzt wird.

Mit `LeseTexturPixel` kann ein Pixel der Textur gelesen werden. Ergebnis ist die Farbe des Pixels. `LeseTexturWert` wird beim Bumpmapping benutzt. Hier wird nicht die Farbe der Textur, sondern die „Höhe" benötigt. Um trotzdem farbige Texturen als Bumpmap benutzen zu können, wird bei `LeseTexturWert` der Schwarzweißwert des entsprechenden Pixels zurückgegeben. Dieser Wert wird anschließend als Höhe interpretiert. Der Schwarzweißwert eines Pixels wird nach der Formel

$$i = 0.3 \cdot r + 0.59 \cdot g + 0.11 \cdot b$$

berechnet. Genau diese Formel wird übrigens auch bei Schwarzweißfernsehern angewendet, um farbige Sendungen entsprechend darzustellen.

Die Berechnung des Schwarzweißwertes erfolgt übrigens schon beim Laden der Textur. Er wird als vierter Wert (zusätzlich zu r, g, b) der Farbtabelle abgespeichert.

Syntax:

```
PROCEDURE LadeTextur(VAR name: FileNameTYP);
```

Parameter:

`name`: Dateiname der Textur im Windows „.BMP"-Format (4 Bit, 8 Bit, unkomprimiert)

Funktionalität:

Die „BMP"-Textur mit dem angegebene Namen wird in den XMS-Speicher geladen, wenn sie nicht bereits gelesen wurde. Aufrufe von `LeseTexturPixel` und `LeseTexturWert` beziehen sich immer auf die letzte mit `LadeTextur` angegebene Textur.

Syntax:

```
PROCEDURE LeseTexturPixel(x, y: INTEGER; VAR r, g, b: BYTE);
```

Parameter:

`x, y`: x- und y-Koordinate des zu lesenden Texturpixels

`r, g, b`: gelesene Farbwerte

Funktionalität:

Liest die Farbe des Pixels (x, y) der aktuellen Textur. Liegen x und y nicht im Bereich der definierten Texturkoordinaten, so werden diese mittels Modulorechnung auf den Bereich abgebildet.

Syntax:
`PROCEDURE LeseTexturWert(x, y: INTEGER; VAR h: BYTE);`

Parameter:

`x, y:`	x- und y-Koordinate des zu lesenden Texturpixels
`h:`	„Höhe" der Textur bei (x, y). Dabei ist die Höhe der Schwarzweißwert des entsprechenden Pixels.

Funktionalität:
Wie bei `LeseTexturPixel`. Statt der Farbe wird der Schwarzweißwert zurückgegeben. `LeseTexturWert` wird beim Bumpmapping benutzt.

C.2.6 Routinen zum Ansprechen von Speicher für polygonale Teilobjekte

Beim Einlesen von Szenen mit Hilfe der Unit `PRIMREAD` werden die Punkt- und Flächenlisten von polygonalen Teilobjekten im XMS-Speicher abgelegt, da unter Umständen die Anzahl der Informationen sehr groß werden kann. Für dieses Abspeichern ist ein globaler Speicherbereich zuständig, der je nach Bedarf automatisch erweitert wird. Soll in diesem globalen Speicher Information abgelegt werden, so muß als erstes mit Hilfe der Routine `NewMem` der nötige Speicher angefordert werden. Als Ergebnis von `NewMem` erhält man eine Adresse. Mit dieser Adresse und der Prozedur `SetMem` kann dann die gewünschte Information abgespeichert werden. Es ist sichergestellt, daß bei einem wiederholten Aufruf von `NewMem` die generierten Adressen „ohne Löcher" hintereinander liegen.

Aus diesem Grund können Sie die Adressen von Informationen des gleichen Typs, die hintereinander mit `NewMem` und `SetMem` gespeichert wurden, mittels

```
(Index der Information)*(Laenge der Information in Byte)
  +(Adresse der ersten Information)
```

berechnen. Mit Hilfe von `GetMem` ist es möglich, diese Informationen dann wieder zu lesen.

Syntax:
`PROCEDURE NewMem(VAR adr: LONGINT; size: WORD);`

Parameter:

`adr:`	Gibt die Adresse des globalen Speichers zurück.
`size:`	In `size` muß die Größe des zu reservierenden Speichers eingetragen werden. `size` muß *gerade* sein, da der XMS-Treiber nur Blöcke mit gerader Anzahl von Bytes verarbeiten kann.

Funktionalität:
Bevor auf den globalen Speicher bei einer bestimmten Adresse mittels `SetMem` zugegriffen werden darf, muß dieser Speicher mit Hilfe von `NewMem` reserviert werden.

Syntax:
`PROCEDURE SetMem(adr: LONGINT; size: WORD; VAR a);`

Parameter:

`adr, size:`	Ziel-„Adresse" und Größe des zu schreibenden Wertes.
`a:`	Zu schreibender Wert (wird nicht verändert)

Funktionalität:
Mit `SetMem` können Variablen in den globalen Speicher kopiert werden. Zu diesem Zweck muß vorher mit `NewMem` der nötige Speicher reserviert worden sein.

Syntax:
`PROCEDURE GetMem(adr: LONGINT; size: WORD; VAR a);`

Parameter:
Die Parameter sind mit denen von `SetMem` identisch.

Funktionalität:
Variablen, die mittels `SetMem` in den globalen Speicher geschrieben wurden, können mit Hilfe von `GetMem` wieder gelesen werden.

C.2.7 Routinen zum Ansprechen der globalen Punkt- und Flächenliste

Außer dem globalen Speicher, der zum Abspeichern von polygonalen Teilobjekten beim Einlesen dient, existieren zwei weitere Speicherbereiche. Der eine nimmt die globalen Punkte, also die dereferenzierten Punkte von polygonalen Teilobjekten auf, der andere die entsprechenden Flächen. Mit Hilfe dieser beiden Listen, der globalen Punkt- und Flächenliste, erfolgt dann die Radiosity-Berechnung und das Rendering mit Hilfe des z-Buffers. Im einzelnen sind folgende Routinen definiert:

Syntax:
`PROCEDURE PunktAnGlobaleListe(VAR p: PunktTYP);`

Parameter:

`p:`	Punkt, der an die globale Punktliste angefügt werden soll. `p` wird nicht verändert.

Funktionalität:
Mit `PunktAnGlobaleListe` kann ein Punkt an die globale Punktliste angefügt werden. Innerhalb der Routine wird die globale Variable `AnzahlDefPunkte` automatisch um 1 erhöht. Die globale Punktliste wird im Initialisierungsteil von `PRIMMEM` auf leer und `AnzahlDefPunkte` auf 0 gesetzt.

Syntax:

```
PROCEDURE PunktInGlobaleListe(Index: LONGINT; VAR p: PunktTYP);
```

Parameter:

Index:	Index des Punktlistenelementes, in das p geschrieben werden soll.
p:	Punkt, der in die Punktliste geschrieben werden soll. p wird nicht verändert.

Funktionalität:

PunktInGlobaleListe schreibt den Wert von p in den Punkt mit entsprechendem Index der globalen Liste. Index darf nicht größer sein als AnzahlDefPunkte, oder, mit anderen Worten: Jedes Listenelement, das mit Hilfe dieser Routine verändert wird, muß vorher mit PunktAnGlobaleListe initalisiert werden. PunktAnGlobaleListe dient also zum ersten Setzen der Punkte in die Punktliste, PunktInGlobaleListe, meist in Zusammenhang mit PunktAusGlobalerListe, zum Verändern der Punktlistenelemente.

Syntax:

```
PROCEDURE PunktAusGlobalerListe(Index: LONGINT; VAR p: PunktTYP);
```

Parameter:

Die Parameter sind identisch mit denen von PunktInGlobaleListe.

Funktionalität:

Mit dieser Routine können Punkte aus der globalen Punktliste gelesen werden.

Syntax:

```
PROCEDURE FlaecheAnGlobaleListe(VAR a: GlobaleFlaecheTYP);
```

Parameter:

a:	Punkt, der an die globale Flächenliste angefügt werden soll. a wird nicht verändert.

Funktionalität:

Diese Prozedur entpricht der Prozedur PunktAnGlobaleListe. Innerhalb der Routine wird die globale Variable AnzahlDefFlaechen automatisch um 1 erhöht. Die globale Flächenliste wird im Initialisierungsteil von PRIMMEM auf leer und AnzahlDefFlaechen auf 0 gesetzt.

Syntax:

```
PROCEDURE FlaecheInGlobaleListe(Index: LONGINT;
    VAR a: GlobaleFlaecheTYP);
```

Parameter:

Wie PunktInGlobaleListe für Flächen.

Funktionalität:

Wie PunktInGlobaleListe für Flächen.

Syntax:
`PROCEDURE FlaecheAusGlobalerListe(Index: LONGINT;` `  VAR a: GlobaleFlaecheTYP);`
Parameter:
Wie `PunktAusGlobalerListe` für Flächen.
Funktionalität:
Wie `PunktAusGlobalerListe` für Flächen.

C.2.8 Routinen zum gezielten Ansprechen von Formfaktoren und unverteilter Energie einer globalen Fläche

Bei der Radiosity-Iteration werden pro Iterationsschritt die Formfaktoren `FF` sowie die unverteilte Energie `dBA` neu bestimmt. Um nun nicht immer alle Bytes der Flächen aus dem XMS-Speicher kopieren zu müssen, existieren zwei spezielle Routinen, die nur auf die angesprochenen Werte einer Fläche zugreifen. Damit dies funktioniert, muß allerdings die Lage von `FF` und `dBA` relativ zum Anfang der Flächeninformationen bekannt sein. Aus diesem Grund wurden der Typ `GlobaleFlaecheTYP` so definiert, daß `dBA` als erstes, direkt gefolgt von `FF` Element des Records ist. Daraus folgt für `dBA` der Offset 0, für `FF` der Offset `sizeof(dBA)`.

Die einzelnen Routinen sind wie folgt definiert:

Syntax:
`PROCEDURE dBAausGlobalerFlaechenListe(Index: INTEGER;` `  VAR dBA: FarbVektorTYP);`
Parameter:
`Index:` Index der Fläche, deren unverteilte Energie gelesen werden soll. `dBA:` Gelesene unverteilte Energie.
Funktionalität:
Mit Hilfe dieser Routine kann gezielt die unverteilte Energie der durch ihren Index angegebenen globalen Fläche gelesen werden.

Syntax:
`PROCEDURE AddiereFFZuGlobalerFlaeche(Index: LONGINT; FF: SINGLE);`
Parameter:
`Index:` Index der Fläche, zu der `FF` addiert werden soll. `FF:` Formfaktor, der zur angegebenen Fläche addiert werden soll.
Funktionalität:
Mit Hilfe dieser Prozedur kann zum Formfaktor der Fläche mit dem angegebenen `Index` der übergebene Werte `FF` addiert werden.

C.3 Die Routinen von PRIMOUT

Die Unit PRIMOUT dient zur Ausgabe der erzeugten Bilder unter PRIM. In ihr sind Routinen vorhanden, die den Bildschirm ansprechen, um auf ihm die erzeugten Grafiken anzuzeigen. Außer PRIMMEM ist PRIMOUT die einzige rechnerabhängige Unit, die von PRIM benutzt wird. Um eine Anpassung auf andere PC-Grafikkarten oder gar andere Rechner als den (IBM-(kompatiblen)) PC leicht zu machen, haben wir versucht, sehr wenige hardwareabhängige Routinen zu benutzen. Dies sind die Routinen `SetzePixel`, `LesePixel`, `GrafikAn` und `GrafikAus`. Alle Routinen in PRIMOUT basieren lediglich auf diesen vier Routinen.

Die auf der Diskette enthaltene Unit PRIMOUT unterstützt diverse Grafikkarten. Im einzelnen sind dies:

- Standard VGA-Karte, 320 × 200, 256 Farben
- VGA-Karte mit TSENG ET 3000, 800 × 600, 256 Farben
- VGA-Karte mit TSENG ET 4000, 1024 × 768, 256 Farben
- SpeedStar 24x (WD-Chip), 640 × 480, 16 Mio. Farben
- SpeedStar 24x (WD-Chip), 800 × 600, 32k Farben
- Spea Video7 Mirage (S3-Chip), 640 × 480, 16 Mio. Farben
- Spea Video7 Mirage (S3-Chip), 800 × 600, 64k Farben
- Genoa 7900 (Tseng ET4000 + TrueColor DAC), 640×480, 16 Mio. Farben

Nicht unterstützte Karten können aber von Ihnen leicht zusätzlich implementiert werden. Für die 256-Farbenmodi erfolgt die Darstellung mittels Dithering nach Bayer (siehe Abschnitt 20.1.4.2), in allen andern Modi (32k Farben, 64k Farben oder 16 Mio. Farben) erfolgt die Darstellung direkt, wobei bei 32k und 64k Farben die unteren Bits der Farben ignoriert werden.

Für die verschiedenen Modi wurden verschieden Pixelsetz- und Pixelleseroutinen implementiert. Näheres hierzu ist in Kap. 20.2 zu finden. Um nun nicht für jedes Pixel mittels eines CASE-Befehls abzufragen, welche Grafikkarte zu bedienen ist, wurden die Routinen `SetzePixel` und `LesePixel` als Prozedurvariablen implementiert. Nachdem mit Hilfe eines Menus einmal eingegeben ist, welche Grafikkarte verwendet wird, erfolgt die Zuweisung der aktuellen Setze- und Leseroutinen an die von PRIM benutzten Prozedurvariablen.

Letztlich können diese Variablen benutzt werden, als wären folgende Prozeduren definiert:

```
PROCEDURE SetzePixel(x, y, r, g, b: INTEGER);
PROCEDURE LesePixel(x, y: INTEGER; VAR r, g, b: BYTE);
```

Übergebene Farbwerte werden innerhalb von `SetzePixel` auf den Bereich 0..255 begrenzt.

Beim Einschalten der Grafik könnnen je nach Grafikkarte unterschiedliche Initialisierungen notwendig sein. Um diese Initialisierungen kartenabhängig vorzunehmen sowie auf einige andere Eigenschaften der Grafikkarte leicht zugreifen zu können, werden Informationen der diversen Karten in einem Feld abgespeichert, das pro Karte einen Eintrag mit folgendem Typ enthält:

```
TYPE GrafikKarteTYP = RECORD
                        Name           : STRING;
                        AufloesungX    : WORD;
                        AufloesungY    : WORD;
                        ExtraInit      : ExtraInitTYP;
                        VModeA         : WORD;
                        VModeB         : WORD;
                        SetzePixelProc : SetzePixelProcTYP;
                        LesePixelProc  : LesePixelProcTYP;
                        BytesProZeile  : LONGINT;
                      END;
```

Hierbei werden folgenden Typen verwendet:

```
TYPE SetzePixelProcTYP = PROCEDURE(x, y, r, g, b: INTEGER);
     LesePixelProcTYP  = PROCEDURE(x,y: INTEGER; VAR r,g,b: BYTE);
     ExtraInitTYP      = (kein,Fuer8BitDither,FuerS3,FuerTseng424);
```

In `Name` ist die Beschreibung der Karte gespeichert. Dieser `Name` wird in dem Auswahlmenu ausgegeben. Die Auflösung ist in `AufloesungX` und `AufloesungY` gespeichert. Die Auflösung der ausgewählten Karte wird in die für `PRIM` globalen Variablen `MaximaleBildschirmBreite` und `MaximaleBildschirmHoehe` kopiert.

`ExtraInit` gibt an, welche speziellen Initialisierungen beim Einschalten des Grafikmodus notwendig sind. In der Routine `GrafikAn` wird dann mit Hilfe einer `CASE`-Verzweigung die entsprechende Extra-Initialisierung ausgewählt und erledigt. In `VModeA` und `VModeB` ist der Videomodus gespeichert. Er wird beim Einschalten der Grafik in den Registern `AX` und `BX` dem Software-Interrupt INT$10 übergeben.

`SetzePixelProc` und `LesePixelProc` sind die für die Grafikkarte spezifischen Prozeduren. Ihr Name kann bereits vor der Definition der Prozedur verwendet werden, wenn die Prozedur als `forward` deklariert wurde.

Um bei Grafikmodi, die sich nur in der Auflösung, nicht aber in der Art und Weise, wie Pixel gelesen oder gesetzt werden, unterscheiden, gleiche Routinen verwenden zu können, ist die Variable `BytesProZeile` definiert, die angibt, wie viele Bytes eine Zeile im Bildschirmspeicher belegt. So kann z.B. für ET3000-Karten mit einer Auflösung von 800×600 (256 Farben) und ET4000-Karten die gleiche Pixelsetzroutine verwendet werden, lediglich die Berechnung der Pixeladresse (und dementsprechend der zu selektierenden Bank) unterscheiden sich dadurch, daß die y-Koordinate mit 800 bzw. 1024 multipliziert werden muß.

`BytesProZeile` ist nicht unbedingt identisch mit der Anzahl der Pixel pro Zeile multipliziert mit der Anzahl Bytes für ein Pixel. So ist z.B. bei S3-Karten im True-Color-Modus bei einer Auflösung von 640×480 eine Zeile nicht 3 Bytes $\times 640 = 1920$ Bytes, sondern 2048 Bytes lang. Dies hat den Vorteil, daß Bankgrenzen mit Zeilenenden zusammenfallen. Bei der SpeedStar 24x ist dies im gleichen Modus anders. Hier kann die Bankgrenze sogar zwischen die verschiedenen Farbanteile eines Pixels fallen, da hier `BytesProZeile` 1920 ist.

Sollen zusätzliche Grafikkarten von PRIMOUT unterstützt werden, so muß

1. das Feld mit den Grafikkarten-Beschreibungen (Variable GrafikKarten) vergrößert und der entsprechende Eintrag definiert werden.
2. Wenn die Karte eine Extra-Initialisierung erfordert, die noch nicht implementiert ist, so muß ExtraInitTYP sowie die CASE-Abfrage in GrafikAn entsprechend erweitert werden.
3. Wenn noch keine Pixelsetz- und Leseroutinen existieren, die für die Karte verwendet werden können, so müssen diese programmiert werden.

Für Grafikkarten im 256-Farben-Modus setzt GrafikAn die Farbpalette so, daß sie für das Dithering verwendet werden kann (3 Bit r, 3 Bit g, 2 Bit b). Hierbei wurde noch ein Trick benutzt, um die Qualität beim Dithern zu erhöhen:

Das Dithern geht davon aus, daß die sichtbare Intensität proportional zum gesetzten Wert ist. Das ist allerdings bei Monitoren i.a. nicht der Fall. Dort ist die Funktion $Intensität = f(gesetzteIntensität)$ in etwa eine Exponentialkurve. Das bedeutet, daß die sichtbare Intensität I_v zu $I_v = I^\gamma$ wird. Um diesen Effekt zu kompensieren, wird in die Farbtabelle nicht eine lineare Folge, sondern eine um dieses γ korrigierte Folge von Farbwerten geschrieben. Der Wert von γ ist von Monitor zu Monitor für r, g und b verschieden.

Eine Routine, die die Farbtabelle mit einer γ-korrigierten Farbrampe in R, G und B setzt, hat in etwa folgende Form:

```
CONST  expr:SINGLE=1.7;  (* Monitor-Gamma rot   *)
       expg:SINGLE=1.7;  (* Monitor-Gamma gruen *)
       expb:SINGLE=1.7;  (* Monitor-Gamma blau  *)
BEGIN                    (* pow berechnet x^y   *)
  FOR i:=0 TO 7 DO
    FOR j:=0 TO 7 DO
      FOR k:=0 TO 3 DO BEGIN
        SetClutEntry(i*32+j*4+k,round(pow(i/7,1/expr)*63)
                                round(pow(j/7,1/expg)*63)
                                round(pow(k/3,1/expb)*63));
      END;
END;
```

Die γ-Werte für r, g und b kann man experimentell ermitteln: Man zeichnet für jeden Farbanteil zwei Rechtecke, wobei das eine Rechteck schachbrettartig mit Pixeln voller Intensiät gefüllt wird, d.h. jedes zweite Pixel wird gesetzt, die übrigen bleiben schwarz. Das zweite Rechteck wird komplett mit dem Tabellenwert für die mittlere Intensität gefüllt. Stimmt das γ, so erscheinen beide Rechtecke gleich hell.

Das eigentliche Dithering wurde bereits in Kap. 20.1.4.2 vorgestellt. Im Gegensatz zur dort abgedruckten Routine, die als Farbwerte Zahlen vom Typ DOUBLE benutzt, die zwischen 0.0 und 1.0 liegen, verwendet die in PRIMOUT implementierte Dithering-Routine ganze Zahlen für r, g und b. Um das Dithering zu beschleunigen, werden bei Initialisierung der Grafik für r, g und b jeweils zwei Felder mit 256 Einträgen berechnet.

Aus diesen Arrays wird dann beim Pixelsetzen mit dem übergebenen Farbanteil als Index der Grundanteil der Farbe am zu bestimmenden Wert und der Vergleichswert für eine 4 × 4-Dithermatrix berechnet. So werden z.B. die Bits des Rotanteils mittels

```
c  := DitherMatrix[x AND 3, y AND 3];
ri := R2Dither[r,0];
IF R2Dither[r,1] > c THEN INC(ri);
```

ermittelt. Hierbei entspricht `r` dem übergebenen Wert, `ri` dem zu setzenden. Beim Lesen eines Pixels wird ein ähnliches Wertefeld verwendet, das im Prinzip invers zu `R2Dither` ist.

`GrafikAus` schließlich schaltet die Grafikkarte wieder in den Textmodus.

Zusätzlich zu den vier bereits erläuterten Prozeduren bzw. Prozedurvariablen sind in `PRIMOUT` folgende Routinen implementiert:

```
PROCEDURE Fehler(s: STRING);
PROCEDURE Linie(x1, y1, x2, y2: INTEGER; r, g, b: INTEGER);
PROCEDURE SetzeRGBAPixel(x, y: INTEGER; r, g, b, a: INTEGER);
PROCEDURE SpeichereFenster(x, y, b, h: INTEGER; fname: STRING;
                           Format: CHAR);
```

`Fehler` gibt die in `s` übergebene Fehlermeldung aus. Zusätzlich wird die Grafik „ausgeschaltet" und das Programm angehalten.

Die Prozedur `Linie` stellt mit einem DDA eine Linie von `(x1,y1)` nach `(x2,y2)` auf dem Bildschirm dar. Dazu wird die in Abschnitt 15.2 beschriebene Festkommadarstellung und die Routine `SetzePixel` verwendet.

Um die Darstellung von transparenten Flächen im z-Buffer zu erleichtern, ist weiterhin eine Prozedur `SetzeRGBAPixel` definiert. Sie bewirkt das Setzen eines Pixels, wobei die Ergebnisfarbe sich aus der gewichteten Summe der neu zu schreibenden und der bereits gespeicherten Farbe zusammensetzt:

```
Farbe := LeseFarbe*(255-a) + NeueFarbe*a.
```

Die letzte Routine, `SpeichereFenster`, bewirkt das Abspeichern eines Teilbildes in eine Datei `fname`. Das Fenster, das gespeichert werden soll, ist durch `x, y, b` und `h` gegeben. Derzeit sind als Formatangaben die Buchstaben „B" für das Windows `.BMP`-Format (24 Bit/Pixel) oder „T" für das Targa `.TGA`-Format möglich. `SpeichereFenster` nutzt die Routine `LesePixel` und kann nur im Zustand „GrafikAn" mit einem dargestellten Bild aufgerufen werden.

Literaturverzeichnis

[ ] *System BIOS for IBM PC's, Compatibles, and EISA Computers.* Addison-Wesley, 2. Auflage.

[AD] Abelson und Di Sessa. *Turtle Geometry.* MIT Press.

[And82] M.H. Andree. Kurvenentwicklung auf Bildschirm und Plotter. *mc*, (5), 1982.

[Arv91] James Arvo (Hrsg.). *Graphics Gems II.* Academic Press, 1991.

[Ber] Berger. *Computer Graphics with Pascal.* The Benjamin/Cummings Publishing Company.

[BFGS86] L. Bergman, H. Fuchs, E. Grant, und S. Spach. Image rendering by adaptive refinement. *ACM Computer Graphics*, 20(4):29–37, August 1986.

[BS81] Bronstein und Semendjajev. *Taschenbuch der Mathematik.* Verlag Harri Deutsch, 19. Auflage, 1981.

[CCWG88] M.F. Cohen, Shenchang Chen, J.R. Wallace, und D.P. Greenberg. A progressive refinement approach to fast radiosity image generation. *ACM Computer Graphics*, 22(4):75–84, 1988.

[CG85] M.F. Cohen und D.P. Greenberg. The hemi-cube: a radiosity solution for complex environments. *ACM Computer Graphics*, 19(3):31–40, Juli 1985.

[Cla90] Ute Claussen. *Verfahren zur schnellen Beleuchtungs- und Schattierungsberechnung.* Dissertation, Universität Tübingen, 1990.

[Cla93] Ute Claussen. *Objektorientiertes Programmieren.* Springer-Verlag, 1993.

[CP91] Ute Claussen und Josef Pöpsel. Der Realität auf der Spur. *c't*, (6,8), 1991.

[CP93] Ute Claussen und Josef Pöpsel. Himmel und Hölle. *c't*, (1):160–170, Januar 1993.

[DK] Bradley Dyck Kliewer. *EGA/VGA: A Programmer's Reference Guide.* McGraw-Hill.

[DSC93] Heiko Duin, Günter Symanzik, und Ute Claussen. *Beleuchtungsverfahren in der Computergrafik.* Springer-Verlag, 1993.

[ES88] J.L. Encarnaçaõ und W. Straßer. *Computer Graphics.* Oldenbourg, 3. Auflage, 1988.

[Fel92] Fellner. *Computergrafik.* BI-Wissenschaftsverlag, 1992.

[fN84] Deutsches Institut für Normung (Hrsg.). *DIN 66 252, Graphisches Kernsystem (GKS).* Beuth Verlag, 1984.

[FvDFH90] J. D. Foley, A. van Dam, S. Feiner, und J. Hughes. *Computer Graphics — Principles and Practice. The System Programming Series*, Addison-Wesley, 2. Auflage, 1990.

[Gla89] Andrew Glassner (Hrsg.). *An Introduction to Ray Tracing.* Academic Press, 1989.

[Gla90] Andrew Glassner (Hrsg.). *Graphics Gems.* Academic Press, 1990.

[Gou71] H. Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers*, C-20(6):623–628, Juni 1971.

[Gri92] I. Grieger. *Grafische Datenverarbeitung.* Springer-Verlag, 1992.

[GTGB84] C. M. Goral, K. E. Torrance, D. P. Greenberg, und B. Battaile. Modeling the interaction of light between diffuse surfaces. *ACM Computer Graphics*, 18(3):213–222, Juli 1984.

[HA90] Paul Haeberli und Kurt Akeley. The accumulation buffer: hardware support for high-quality rendering. *ACM Computer Graphics*, 24(4), August 1990.

[Hal88] Roy Hall. *Illumination and Color in Computer Generated Imagery. Monographs in Visual Communications*, Springer-Verlag, 1988.

[Har88] Steven Harrington. *Computergrafik – Einführung durch Programmierung.* McGraw-Hill, 1988.

[HP89] Ch. Hornung und J. Pöpsel. 3D á la carte. *c't*, (4–10), 1989.

[KG79] D.S. Kay und D. Greenberg. Transparency for synthesized images. *ACM Computer Graphics*, 13(2):158–164, August 1979.

[Kir92] David Kirk (Hrsg.). *Graphics Gems III.* Academic Press, 1992.

[KPRR91] Bert Klöppel, Manfred Paul, Jürgen Rauch, und Jochen Ruhland. *Compilerbau – Am Beispiel der Programmiersprache SIMPL.* Vogel-Verlag, 1991.

[LKV90] Olin Lathrop, David Kirk, und Doug Voorhies. Accurate rendering by subpixel addressing. *IEEE Computer Graphics & Applications*, 45–53, September 1990.

[LMS91] Wolfgang Leister, Heinrich Müller, und Achim Stößer. *Fotorealistische Computeranimation.* Springer-Verlag, 1991.

[Mar92] Neal Margulis. *Der i860 – Architektur und Programmierung.* tewi-Verlag, 1992.

[Mic] *Microsoft Windows Software Development Kit.*

[NS81] W.M. Newman und R.F. Sproull. *Principles of Interactive Computer Graphics.* McGraw-Hill, 1981.

[Per85] Ken Perlin. An image synthesizer. *ACM Computer Graphics*, (3):287–296, 1985.

[PH90] J. Pöpsel und Ch. Hornung. Highlight shading: lighting and shading in a PHIGS+/PEX environment. *Computers & Graphics*, 14(1), 1990.

[Pho75] B.T. Phong. Illumination for computer generated images. *Communications of the ACM*, 18(6):311–317, Juni 1975.

[Pla87] Jürgen Plate. *Computergrafik.* Franzis Verlag, 1987.

[Ple] Axel Plenge. *3D-Grafik und Animation.* Markt & Technik.

[Pur86] W. Purgathofer. *Graphische Datenverarbeitung.* Springer-Verlag, 1986.

[SSS74] I.E. Sutherland, R.F. Sproull, und R.A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[Ste90] *Stereographics.* Course Notes SIGGRAPH Dallas, 1990.

[Ups89] Steve Upstill. *The RenderMan Companion.* Addison-Wesley, 1989.

[vD88] A. van Dam. PHIGS+ functional description revision 3.0. *ACM Computer Graphics*, 22(3):125–218, Juli 1988.

[WCG87] J.R. Wallace, M.F. Cohen, und D.P. Greenberg. A two–pass solution to the rendering equation: a synthesis of ray tracing and radiosity method. *ACM Computer Graphics*, 21(4):311–320, Juli 1987.

[Whi80] T. Whitted. An improved illumination model for shaded display. *Communications of the ACM*, 23(6):343–349, Juni 1980.

[Wir81] Niklaus Wirth. *Compilerbau.* Teubner-Verlag, 1981.

[WW92] Alan Watt und Mark Watt. *Advanced Animation and Rendering Techniques – Theory and Practice.* Addison-Wesley, 1992.

Index

Im folgenden Verzeichnis werden zwei Arten von Einträgen unterschieden. Einträge, die im normalen Schriftstil gehalten sind, bezeichnen Begriffe, die im Text erläutert bzw. erwähnt werden. Ein Beispiel hierfür ist das Wort „Animation". Stichworte, die in Schreibmaschinenschrift dargestellt sind, stehen für Worte der Eingabesprache, z.B. „`AUGE`", oder Bezeichner aus der Software, z.B. „`BerechnedFFs`".

Abbildung
- affine 12
- bijektive 12
- zweidimensionale 13

Abklingkoeffizienten 109
Absorption 95
Abtastung 246
- Über- 72
- statistische 249

Achsenspiegelung 13
`AddiereFFZuGlobalerFlaeche` 380
Algorithmus 62
- CSG- 243
- Energieberechnung 186
- Flächendarstellung 147
- Flächenfüllen 25, 70
  rekursives 25
- Formfaktorberechnung 185
- Gauß- 166
- Gouraud- 116, 128
- Konstanter Schattierungs- 127
- Maler- 68
- Mehrpaß 160
- modifizierter Bresenham- 253
- Phong- 128
- Punktfarbenberechnung 188
- Raytracing- 201
- Rendering-Pipeline 86
- Scanline- 69
- Singleplane- 184
- Tiefensortier- 68
- Umwandlung von Flächen 174
- Umwandlung von Patches 175
- von Cohen und Sutherland 66
- von Sutherland und Hodgman 66
- z-Buffer- 68, 113, 114

Alias-Effekt 71
Aliasing 71, 158, 246
`AllokiereIBuffer` 373
`AllokiereZBuffer` 371
α-Buffer 121
`AMBIENTELQ` 108, 342
`AmbienteLqTYP` 106
Animation 154, 156, 251, 255
- Farbpaletten- 266
- Geschwindigkeit 268

Antialiasing 71, 247
Approximation 64
- Bézier- 36
- exponentielle 32
- lineare 30
- logarithmische 32

Archimedische Spirale 29
`AUGE` 341

backface culling 82
`Beleuchte` 105, 107
Beleuchtung 57, 89
- Berechnung 69, 84

Beleuchtungsmodell 89, 116
- Implementierung 105

Beobachterstandpunkt 195
`BerechnedFFs` 181
`BerechneFlaechenGroessenUndAmb` 180

BerechneFormfaktoren 184
BerechnePunktFarbenAusEnergien 187
Beschleunigung
– Grafik- 305
BestimmeMatrizenFuerSingleplane 182
BetrachterTYP 348
Bewegungsunschärfe 251
BGI 282
BILDABSTAND 341
bildaus 276
BILDBREITE 341
Bildebene 341
Bildschirm
– Koordinaten 58, 82
– Seite 296
– Steuerung 259
BILDSCHIRMFENSTER 341
Bildspeicher 259, 279
Binomialverteilung 37
BIOS 264
Blickpunkt
– Koordinaten 59, 61
Blickreferenzpunkt 60
bounding box 252
BPD 61
Brechung 199
Brechungsgesetz 96, 199
Brechungsindex 148, 200, 339
Bumpmapping 119

CallXMS 368
CGA 261
CISC 314
Clipping 21
– Algorithmus 21
– Code 84
– Cohen-Sutherland 23
– einfaches 21
– Linien 21
– Polygone 24
CMY-Modell 90
colorrec 195
conerec 219
CSG 63, 239, 330
– Baum 254

Darstellung
– monochrome 259
– Stereo- 295
Datenstrukturen 345
dBAausGlobalerFlaechenListe 380
DDA 70
– unsymmetrischer 70
Dereferenzieren 358
DereferenziereSzene 357
Diagramm
– Roth- 241
Differentiation 31
distributed ray tracing 249
DITHER1 273, 275
DITHER2 277
Dithering 270
– Bayer- 270
– Floyd-Steinberg 275
– Matrix 271
rekursive 272
Doppelkegel 216
doraytrace 196
DREHE 338
drehetrans 296
Drehung 13, 16, 18
Dreieck
– Flächeninhalt 181

Ebene 229
– Polarisations- 300
Ebenengleichung 230
EGA 262
EigenschaftTYP 106, 145, 352
Eingabesprache 58, 325
Einzelschrittverfahren 166
Element 170, 173
ElementeAufSingleplane 183
Ellipsoid 214
Energie 160, 167
– Rest- 167
– unverteilte 380
– verschießen 170
Energiedichte 144, 156
– ausgestrahlte 187
– empfangene 187
enter_cone 220
error diffusion 275
ErwEigenschaftTYP 352
ErweitereXMSSpeicher 370
ErwFlaechenTYP 349
ErzeugePatchesUndElemente 173
ET4000 283

FACET 333
Facettierung 173
Farbanteil 153
Farbe 95
– diffuse 144, 339
– spiegelnde 144, 339
Farbgewichtung 144
Farbmodell 89
Farbpalette 261
– Animation 266
Fehler 384
Fehlerfortpflanzung 277
Fenstergranularität 287
Festkommadarstellung 129
Filter 37, 72
FlaecheAnGlobaleListe 379
FlaecheAusGlobalerListe 379
FlaecheInGlobaleListe 379
FlaecheTYP 349
flat shading 116
Flat-Shading 127
FLIP 332
Fläche 25, 63, 378
– diffuse 154
– füllen 24
– Lichtquelle 93
– Nachbar- 177, 179
– Unter- 87
Flüssigkristallfeld 299
Formfaktor 156, 165
– Berechnung 158, 169, 185
– Teil- 181, 375
Fotorealismus 154
FPD 61
FRACTINT 296
frame buffer 259
full matrix 165
Funktion
– stetige 29

γ-Korrektur 383
Gamma-Korrektur 383
GebeIBufferFrei 373
GebeZBufferFrei 371, 372
GED 58, 325, 361
– Betrachtungsstandpunkt 341
– CSG 330
– Datei 328
– Fläche
Eigenschaften 338
– Kegel 336
– Körper
vordefinierter 334
– Lichtquelle 342, 348
– Molekülring 58
– Objekt 327
– Polygon 329, 331
– Torus 336
– Transformation 330, 338
– Version 328
– Würfel 334
– Zylinder 336
GeometrieDatenTYP 347
GerichteteLq 108, 342
GerichteteLqTYP 106
Gerätekoordinaten 10
Gesamt
– Energie 167
– Transformation 19
Gesamtschrittverfahren 166
Gesetz
– Brechungs- 96
– Kosinus- 97
– Reflexions- 96
– von Lambert 97
GetMem 378
Gleichungssystem 159, 165
GlobaleFlaecheTYP 144, 354
Gültigkeitsbereich 340
Gouraud-Shading 128
GrafikAn 381
GrafikAus 381
Grafikkarten 259
– PC- 279
GrafikKarteTYP 382
Grenzfläche 95

Halbschatten 249
Halbtontechnik 268
Hardware 259
Helligkeitsmatrix 275
Hemicube 157
HiColor 291
highlight shading 117
HINTERCLIPEBENE 341
Hintergrund 192, 328, 342, 348
HINTERGRUND 342
HintergrundTYP 348

Hinterschneiden 26
Histogramm 30
hit_disc 231
hit_plane 231
HLS-Modell 90
Hornerschema 34
HSV-Modell 90
Hyperboloid 215
Hypozykloide 30

i860
- Aufbau 314
- Befehle 317
- Befehls-Cache 317
- Bildspeicher 321
- Daten-Cache 317
- Fließkommaeinheit 316
- Grafikeinheit 316
- Kerneinheit 314
- Register 317
IEEE-Standard 316
Index
- Brechungs- 148
- Flächen- 177
- Punkt- 145, 350
initkoor 296
initpalette 273
initsincos 296
InitXMS 368
Integration
- numerische 71
Intensität 90, 93
- Abschwächung der 93
- Berechnung 97
Interpolation 64, 129
- der Normalenvektoren 116
- lineare Farb- 116
- Phong- 116
- Polynom- 33
- Spline- 34, 35
intersect 201, 243
inter_cone 222
inter_sphere 210
item-Buffer 373, 374
Iterationstiefe 196

KA 338
KD 338
Kegel 216
KEGEL 336
Kegelstumpf 219
- massiver 227
Körper 207
- Rotations- 237
- vordefinierter 329
Konzentrationsexponent 92
Koordinaten 9, 48, 57, 62
- Bildschirm- 58
- Blickpunkt- 59
- homogene 18, 48, 49
- kartesische 12
- Modell- 57
- normierte 58
- normierte Projektions- 61
- Textur- 118
- Welt- 57
Kopieren
- Speicher 370
Kosinusgesetz 97
KS 338
Kugel 207
- umhüllende 254
KUGEL 334
Kurve 36

LadeTextur 376
LCD-Shutter 38
LCD-Shutter-Brille 299
LesedFFBufferZeile 375
LeseGEDDatei 356
LeseIBufferZeile 374
LesePixel 381
LesePixelProcTYP 382
LeseTexturPixel 376
LeseTexturWert 376
LeseZBufferZeile 372
LeseZIBufferZeile 374
Lichtpunktmethode 301
Lichtquelle 90, 191, 342, 348
- ambiente 91
  Implementierung 108
- Flächen-
  homogene 93
- Modell 89
- Punkt- 92
  Implementierung 109
- Richtungs- 92
  Implementierung 108

– Vektor 98
LichtquellenTYP 107
Lichtstrahl 191, 194
– Darstellung 194
Lichtvektor 197
Linie 384
Linien 69
LoescheGesamtenIBuffer 373
LoescheGesamtenZBuffer 372

Mach-Band-Effekt 116
Matrix 43
– adjunkte 166
– diagonaldominante 159
– Dither- 271
– Helligkeits- 275
– Implementierung 79
– inverse 44, 81
– Multiplikation 45
– transponierte 43
MittlereBeteiligteFlaechen 187
Modell
– Beleuchtungs- 89
– Farb- 89
CMY- 90
HLS- 90
HSV- 90
RGB- 90
Modellierung 58, 62
– flächenorientierte 63
– volumenorientierte 63
Modellkoordinaten 57
Moiré-Muster 270
motion blur 251
mySetblockRGB 265
mySetRGBPalette 265

N 339
Nachbarschaften
– Flächen- 175
NewMem 377
Ähnlichkeitsverhältnis 13
Überabtastung 246
Überdecken 23
Normale 145
Normalenvektor 197
– Berechnung 116
– interpolierter 237
– Modulation 119
normal_cone 228
normal_sphere 211
notinshadow 201
NRP 60
Nusselt-Analogon 157

Oberflächeneigenschaft 352
– Implementierung 106
Objekt
– Referenz auf 329
– Teil- 328
– vordefiniertes 329
ObjektTYP 347
OD 339
OE 339
ordered dither 270
OS 339

PALANIM 267
PALETTE 266
Parallelisierung 255
Parameterdarstellung 29, 63
Patch 170, 173
PC-Grafikkarten 279
Perspektive 195
PHIGS PLUS 90
Phong-Shading 116, 128, 148
Pipeline
– Flächen- 84
– Punkt- 84
Pixel 69, 81
– Sub- 71
Pixelfarbe 148
PIXELRATIO 341
Polarisation 300
Polarkoordinaten 16
Polygon 69, 329, 331, 349, 377
– Schattierung 117
POP 340
Portierung 52
PRIM 52
PRIMDERE 345, 356
PRIMGLOB 79, 345
PRIMMEM 52, 79, 371
PRIMOUT 52, 79, 381
PRIMRAD 172
PRIMREAD 325, 345, 356
PRIMSHD 142
Präsentationsgrafik 30

Programm
– Bedienung 53
progressive radiosity 165
Projektion 48
– Koordinatensystem 61
– Parallel- 48
– perspektivische 49
– Referenzpunkt 61
PRP 61
Punkt 41, 378
– Farbe 144
– Lichtquelle 92
– scharfer 84, 143
`PunktAnGlobaleListe` 378
`PunktAusGlobalerListe` 379
`PunktInGlobaleListe` 378
`PunktLq` 109, 342
`PunktLqTYP` 106
`PunktTYP` 106, 145, 349
`PUSH` 340
`pxet4000` 284
`PXHICOL` 293
`PXVESA` 291

Quadriken 213

Radiosity 57, 153
– Hauptprogramm 169
– Iteration 159
– Lösung 159
  iterative 166
– Verfahren 167
  Ablauf 165
Raumeinteilung 253
`rayrec` 195
Raytracing 57, 153, 191
– Beschleunigung 252
– rekursives 193
– verteiltes 249
`RDKRAY` 195
`REFERENZPUNKT` 341
`Reflektivitaet` 145
Reflektivität 144, 187
– mittlere 167
Reflexion 95
– ambienten Lichtes 96
– diffuse 97
– spiegelnde 97, 98
Reflexionsgesetz 96
Reflexionskoeffizient 97
– ambienter 338
– diffuser 338
– spiegelnder 338
Reflexionsvektor 197
Rendering 57
– auf die Singleplane 182
– Pipeline 57, 82
Restenergie 167
RGB-Modell 90
Richtungslichtquellen 92
Rückfläche 65, 82
– Entfernung der 65
Rot-Blau-Technik 295, 296
Rot-Grün-Brille 295
Rotation 46
Rotationskörper 237
`ROTGRUEN` 296

Scankonvertierung 69, 114
– von Dreiecken 115
Schatten 192, 197
– Halb- 249
Schattierung 69, 115, 128
– Art der 339
– Dreiecks- 116
– konstante 116
– Parameter 143
– Polygon- 117
Schnittmenge 240
Schnittpunkt 208
Schnittpunktberechnung 63
– mit einem Dreieck 235
– mit einem Polygon 236
– mit einer Ebene 229
– mit einer Kreisfläche 231
– mit einer Kugel 207
– mit konvexem Viereck 232
Schraffieren 24, 26
`SchreibedFFBufferZeile` 375
`SchreibeZBufferZeile` 373
`SchreibeZIBufferZeile` 374
Segment 283
`setactivepage` 297
`setdot` 273
`setdotohnedither` 274
`setetpixel` 284
`SetMem` 377
`setnewmode` 281, 284

setpixel 274
SetRGBPalette 264
setvesapixel 291
setvisualpage 297
SetzeDitherPixel 271
SetzePixel 381
SetzePixelProcTYP 382
SetzeRGBAPixel 384
SetzeSchattierungsParameter 143
shade 203
Singleplane 158, 169, 171, 182
Skalarprodukt 47, 203
SKALIERE 338
Skalierung 16, 18, 46
smooth scrolling 290
Säulengrafik 30
SolidAusdruckTYP 351
SolidDefinitionTYP 351
SolidTermTYP 351
Span 69, 70
speculardirection 201
SpeichereFenster 384
Speicherverwaltung 367
Spiegelachse 13
Spiegelung 15, 18, 197
Spline
– kubischer 34
Spotlichtquelle 92
Sprungstellen 29
StelleFlaecheDar 144, 147
STEREO 303
stereo1 304
stochastic sampling 249
STRAHLER 110, 342
StrahlerTYP 106
Strahlverfolgung 153, 156
Subpixel 71
SuchePatchMitGroesstemdBA 181
Super-VGA 279
Supersampling 246
– adaptives 247
Superscalar-Mode 314
Synchronimpulse 260
Synchronisation 303
Szene 165
SZENE 327

T 339
T-Stück 177
Teilformfaktor 375
Teilobjekt 328, 349
TeilObjektTYP 347
Term
– ambienter 160, 167
Textur 117, 148, 212, 216, 339, 375
– Arten 118
– Bump- 120
– Diskretisierung 119
– dreidimensionale 118
– endliche 119
– Farb- 118
– Volumen- 120
– zweidimensionale 118
TEXTUR 145
texture mapping 118
Texturkoordinaten 118
TexturTYP 352
Texturwert
– Berechnung 118
Thermodynamik 154
Tortengrafik 30
TORUS 336
Totalreflexion 200
trace 196, 202, 206
Transformation 45, 47, 57, 59, 330
– Projektion 48
– Rotation 46
– Skalierung 46
– Translation 46
– Verkettung 330
– von Koordinatensystemen 48
– von Normalenvektoren 47
Translation 13, 16, 19, 46
Transmission 95
– diffuse 97
– spiegelnde 97
transmissiondirection 202
Transparenz 121, 148, 339
Tridiagonalmatrix 35

Unterteilung
– Kanten- 177
UPVEKTOR 341

Vekcomb 202
Vektor 41, 43
– Licht- 197
– Lichtquellen- 98

– Normalen- 47, 197
– Reflexions- 197
– Spalten- 41, 45
– Zeilen- 41, 45
vektor 195
Verdeckung 67
– objektorientierte 68
– pixelorientierte 67
Verdeckungsrechnung 67, 113
Verfahren
– adaptive 117
– Floyd-Steinberg- 275
– Newton- 33
– Roth- 239
VERSCHIEBE 338
VESA 282, 285
VGA 261
Viereck
– planares 232
Viewport 10
viewrec 195
Volumenmodell 63
VORDERCLIPEBENE 341
VPD 61
VPW 61
VRAM 261
VRC 60
VRP 60
VUP 59
WandeleFormfaktorenInEnergieUm 185
Weltkoordinaten 10, 57
Wertepaar 30
Window 10
WUERFEL 334

XMS
– Einsprungadresse 368
– Speicher 349, 367
– Treiber 52, 368
XMS_ParameterBlockTYP 370

z-Buffer 113, 149, 371, 374
Zahl
– Festkomma- 129
ZBufferTest 373
ZeichneDreieck 148
ZeichneHintergrund 149
ZeigeIBuffer 374
zeile0aus 277
Zylinder
– elliptischer 217
ZYLINDER 336

_3TYP 107

Springer-Verlag und Umwelt

Als internationaler wissenschaftlicher Verlag sind wir uns unserer besonderen Verpflichtung der Umwelt gegenüber bewußt und beziehen umweltorientierte Grundsätze in Unternehmensentscheidungen mit ein.

Von unseren Geschäftspartnern (Druckereien, Papierfabriken, Verpakkungsherstellern usw.) verlangen wir, daß sie sowohl beim Herstellungsprozeß selbst als auch beim Einsatz der zur Verwendung kommenden Materialien ökologische Gesichtspunkte berücksichtigen.

Das für dieses Buch verwendete Papier ist aus chlorfrei bzw. chlorarm hergestelltem Zellstoff gefertigt und im pH-Wert neutral.